

TRAILBLAZER

A new architecture for Rails.



Nick Sutterer

Trailblazer

A New Architecture For Rails

Nick Sutterer

This book is for sale at <http://leanpub.com/trailblazer>

This version was published on 2015-12-14



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

©2014 - 2015 Nick Sutterer

Tweet This Book!

Please help Nick Sutterer by spreading the word about this book on [Twitter](#)!

The suggested hashtag for this book is [#trbrb](#).

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

<https://twitter.com/search?q=#trbrb>

To my parents.

To my Madre who never gets tired of teaching me the simplicity of cooking, and, to my Dad who never gets tired of teaching me the importance of believing in your ideas.

Contents

Introduction	1
How To Read This Book	2
The Missing Architecture	3
Learnings About Reality	4
The Trailblazer Architectural Style	6
Why Trailblazer?	6
A Non Intrusive Framework	7
Trailblazer In A Nutshell	10
Logicless Models	11
Concepts and File Structure	12
High-Level Domain	13
Operation	15
High-Level Architecture	18
Validations	19
Inheritance Over Configuration	23
Builders	25
Authorization And Policy	26
Authentication	28
Representers	29
API: Parsing and Rendering	30
Using Hypermedia	32
Rendering Views	33
Cells	34
Twin	37
Testing	38
A Note On Complexity	40
Summary	40
Operations And Forms	42
The Example app	42
The Create Operation	44
Controllers	48
Reform: Form Objects	51

CONTENTS

Operation's Form Integration	53
Model Semantics	54
Rendering Forms	56
Testing Validations	57
Testing Controllers: Integration Tests	59
Update Operation	60
Reform and <code>strong_params</code>	64
Cells	66
Rails Views and Encapsulation	66
View Models	67
Modelling the UI with Cells	68
Anatomy of a Cell	69
Rendering Collections	73
Integration Tests	76
Cell Tests	77
Nesting Cells	78
Rails and MVC	79
Final Test Setup	80
The <code>cell</code> Helper	81
Summary	81
Nested Forms	82
Adding Comments	82
The Comment Concept	85
The <code>setup_model!</code> Hook	86
Nested Contracts	87
Pre-populating Forms	89
Form Processing	91
Populating Forms for Validation	92
Saving Nested Objects	95
Flash messages and Redirecting	96
Readers for Operations	96
Static Form Population	97
Form Presentation Helpers	98
Pre-selecting Values	99
Operation and Form Tests	100
Composed Views	102
The <code>present</code> Helper	102
Multiple Operations for Composed Pages	105
Form Submission: Widgets vs. "RESTful"	106
Writing a Cells Feature	109

CONTENTS

Application-wide Features	110
Simple Decorator Helpers	112
Rails' View Architecture	113
Kaminari Pagination and Cells	115
AJAX Pagination with Cells	118
On Controller Structuring	121
Cells Tests	122
Smoke Tests	125
Summary	127
Mastering Forms	128
Things and Authors	128
Adding Authors	129
Dynamic Prepopulation	130
Validation Population	131
Skipping Blanks	132
Form Debugging Techniques	133
Saving	135
Adding Authorships	136
Non-CRUD Behavior	138
Complex Validations	138
Testing Create	140
Presentation Tests	142
Updating Things	143
Hacking the View	143
Helpers in Forms	144
Form Inheritance	145
Virtual Properties	146
Populator	147
Persisting the Deletion	150
Testing Update	151
Extracting Forms to Separate Files	153
Callbacks	156
Domain Callbacks	157
The Persisted Module	160
Explicit Callbacks	162
Imperative Callbacks	164
Callbacks in Operations	167
Testing Callbacks	170
View Caching	171
Cache Keys	172
Expiring Keys	173

CONTENTS

Debugging View Caching	174
Caching Composed Views	175
The CacheVersion Pattern	176
File Uploads	179
Paperdragon	180
File Validations	182
Callback Groups	183
Rendering Images	184
Testing Uploads	185
Conclusion	187
Authentication	188
Populating by ID	188
Tyrant	190
Sign Up	192
Authenticatable	196
SignUp Form	199
Testing SignUp	200
Sign In	201
Modelless Forms	202
Login	203
Application-wide Tyrant	205
A Warm Greeting	207
Signing Out	208
Testing Logins	209
Putting Users to Sleep	210
Waking Up Sleeping Users	212
Integration Tests for Wake Up	217
Authorization And Polymorphism	219
Policies	219
Operation Inheritance	221
Polymorphic Builders	223
Resolver	224
Refining Operations with Modules	226
Polymorphic Testing	227
Polymorphic Views	230
Integration Test: Create	233
Updating	236
Composable Operations	237
Update Tests	240
Delete	245
Impersonate	249

CONTENTS

Hypermedia APIs: Rendering	254
File Structure	254
Rendering Comments	255
HAL Hypermedia Links	256
Rendering	257
Representer	258
GET Controller	260
Responders	261
Inferring Representer	262
Composing Representer	263
Rendering Nested Documents	265
Scopes and Sorting	267
Index Operation	269
Lonely Collections	271
Filtering	272
Hypermedia APIs: Deserialization	276
Deserialization in Trailblazer	277
Parsing Comment	278
Deserialization and Contract	281
Deserialization in Reform	282
Deserializing HAL	283
Create	285
Polymorphic Update	289
Basic Auth	290
Discussion: Polymorphic Operations and APIs	294
Perspectives	295
This Book	296
Why Not a New Framework?	296
Rails	297
Webmachine	298
Grape	298
ROM	298
Roda	298
Lotus	299
Trailblazer	299

Introduction

In the late summer of 2006 I had my first date with Rails. Oh, I remember it, I was nervous and had sweaty palms. Tobi, a good friend of mine, walked me through models, views, controllers and tests. He was so excited about it that he could already recall two dozens of TextMate shortcuts for arbitrary things you apparently needed for writing Rails apps.

Coming from Perl and PHP, both used with a strict and very object-oriented discipline, but both incredibly ugly languages to look at, I was fascinated by Ruby and its beautiful syntax. And I loved ActiveRecord! I had been working on a object-relational mapper in Perl myself for years, its name *DataObjects*, its implementation far away from ActiveRecord's maturity.

And I remember digging the integration of tests in Rails. Tests have been an integral part of my work before Rails, but it was always a bit messy and felt clumsy. Now, a framework that ships with automatic testing - amazing!

So I set up a basic Rails app for tracking how many beers I was having at parties and to calculate my blood-alcohol value at every time of the night. Well, that was the goal, and I'm still working on it, haha. Ridiculous, I know.

After about two hours I had models and first unit tests in place and I was just enjoying the simplicity how SQL was abstracted and how controllers would let me create dynamic web pages without all that pain.

While I'd still create users via the console, I wanted to display a small box in several pages, showing the users who recently joined the project, a bit like a sidebar. Even though there wasn't even a form to sign up, yet, I found this an important feature.

And this was the birth of Trailblazer.

Of course, I didn't know that by the time.

I literally spent days to find out how to encapsulate a piece of Ruby logic along with a part of a view in Rails. Forums (yeah, back then, we used forums), IRC chats (yeah, back then we were using "chat systems") and talking to some fellow Rails developers in town (yeah, back then we used to actually *talk* to each other in real life) didn't help.

Everyone kept telling me to use a partial, a shared controller action, a shared `before_filter` and a helper. And people told me that this is perfect and exactly how it's done and everything else is just a waste of time and no one needs more encapsulation.

That's what they said, back then.

I wasn't happy with that at all. It felt wrong to spread code of a single component across all the layers of the framework.

Eventually, I'd stumble upon a small Ruby script called `cells.rb`, written by the great Ezra Zygmuntowicz¹. I never had the chance to say Thanks to Ezra in person for these few lines of code he'd written, but they have changed my life.

`Cells` was a little script that provided plain Ruby classes with the ability to use the Rails rendering stack - exactly what I wanted!

Cells became a Rails plugin, then a Ruby gem, then moved to Github, and it has kept my life busy with programming for years. Of course, it wasn't only Cells.

Once I was in that mindset of questioning Rails and its architectural concepts, I couldn't stop. It was too late. Everytime I ran into a problem with Rails, I instantly started writing a gem to solve it. Every gem grew its own ecosystem and users soon started to use other gems from my alternative stack, too, which made me work even harder on integrations between the different layers.

Many years, ten-thousands of lines of code written, tested, deleted, a few dozens Ruby conferences and hundreds of beers later, I present you Trailblazer.

This is my distill, my extraction from almost a decade in the Ruby and Rails community, a mesh-up of the gems I found helpful and an attempt to introduce desperately needed abstraction layers into this great framework.

Enjoy the ride, and never forget: Hydration is key.

How To Read This Book

You might wonder how to read this book, an my answer is "*In a cosy place with a coffee or a beer at hand.*" All the code to understand the concepts is printed in the book, so you not necessarily need a computer. However, it is helpful to browse the tagged versions for additional code. For example, I don't discuss every single test case I wrote.

I spent a remarkable amount of time structuring this book. The structure aims to allow you to skim over or skip entire chapters. Even if you don't know how forms exactly work or what's a twin's job you will be able to understand it from the context. So, what I'm saying is, I encourage you to jump to chapters that attract you most. Although not harmful, there's no need to read this book from the first page to the last.

Also, please take breaks. The chapters are designed to roughly take you 45 minutes to read and to flick through the corresponding code on Github. At the beginning of each chapter I will link to a branch on the [gemgem-trbrb²](#) repository which contains all the code of the running application at that very state of the book.

Chapter 2, *The Trailblazer Architectural Style*, is a comprehensive summary of the patterns and concepts used throughout this book. It aims to be a formulation of the architectural style with

¹RIP, brother.

²<https://github.com/apotonick/gemgem-trbrb>

minimal code examples. All following chapters contain a lot of code where we actually use Trailblazer to build a real application.

I advise you to jump directly into chapter 3, *Operations And Forms*, for now! Chapter 2 is a very interesting, but also a bit longer read, and might be even more fascinating after you've played with Trailblazer a bit.

The Missing Architecture

For every Rails project, there is exactly two outcomes. Either someone in the team's an experienced architect and leads the software to an advanced design with service layer, view components, maybe forms, and so on. Or, and that's the classic way, the project strictly follows the *Rails Way* and will end up as a code disaster.

Explaining why a conventional Rails architecture fails is simple: There *is* no architecture.

The fatal delusion that three abstraction layers, called "MVC", are sufficient for implementing complex applications is failure by design.

This is not because Rails is "Omakase" and everyone is supposed to mix in what they feel is right, no, that is not the purpose of a framework! A framework's job is to provide boilerplate code and architectural guidance. Claiming that Rails is "Omakase" is an excuse for the complete lack of the latter.

Shortly after its inception in 2004 the Rails core team has completely stopped innovation on the architectural level. Probably assuming that the degree of abstraction is good enough, an incredible work load has been put into making it faster, cleaner, less coupled, adding fantastic features to ActiveRecord, a master-piece of software craftsmanship, integrating it with other products like test frameworks or CoffeeScript - the list could go on for pages.

And now don't get me wrong. I couldn't express how much respect I have for all those people and all the work that has been done on Rails and its ecosystem. It is beyond my imagination how many hours have been spent on all that, and that mostly because people believe in Open Source and want to share benefits of their work with others.

What made me work on an alternative stack for Rails is the realization that the core team doesn't want change. The fact that thousands of projects that had a quick start with Rails are now in an unmaintainable state of chaos bubbles up the *Rails Ivory Tower™* only slowly, like the hang-over when brushing your teeth the next morning after a wild party night without decent hydration.

On an architectural level, everything in Rails is crying for separation and encapsulation. But those cries go unheard, mostly.

Instead of identifying and implementing new layers like form objects, things get violently pressed into the controller and the model, both in the framework itself and in your application code. All that just because of fearing over-abstraction. Why on earth is Rails constantly trying to solve incredible complex problems in one gigantic, monolithic class?

Learnings About Reality

When I was younger I tended to stand up for hard-core encapsulation. When it came to Rails, I saw the problems of the monolithic design and I wanted to solve all the dependencies by separating things in a very strict way. While I still believe in separation of concerns and encapsulation more than ever, I've learned an important lesson.

The reality is, people use Rails because it helps them shipping products, getting stuff done in a very fast way. It takes a few hours and you might have a first presentable version of a shop, for instance. This is a result of zero encapsulation in Rails. You just build things without thinking too much about where goes what. This is why Rails is so popular: you knock together a project with a minimum of conventions, interfaces and specifications. And this is awesome!

Of course, a software project of any size takes several months to get production-ready. It will grow and the architecture - or better: the lack of architecture - will become a problem at one point. This is when people realize the down-side of the *Rails Way*. This is when you start visualizing your application as a hairy, growling mountain creature that might attack and eat you if you don't watch out.

I've learned to understand this desire to ship stuff. It is extremely motivating to get something up and running in days. And a nice side effect: you make money faster than your competitors. I've also learned to never give up architecture in trade for shipping. This will always end in the aforementioned disaster. Sustainability is my maxim when writing code.

Now, what am I trying to say? I believe the puristic *Rails Way* isn't appropriate for projects with a complexity greater than a 5-minute blog. Full stop. I also believe that writing an application with J2E specifications sucks. I don't want to be constantly thinking about types and interfaces and builders and how to wire them together.

Of course, my last statement is a provocative one.

That Rails actually is capable of running real world apps has been proven by thousands of production apps out there. Nevertheless, the question is: what price do we pay for Rails' *simplicity* in trade for *Maintainability*?

Is it worth writing models with thousands of lines of code and building a throw-away software kludge that is impossible to maintain, just for the sake of development speed? Is it worth blindly hacking away and following the Rails conventions without thinking about whether this is the right place for that code?

On the other hand: is it worth to introduce a high level of framework complexity and to constantly question your code design? Is it worth sacrificing time for a clean, well-designed system?

As far as I can see, Trailblazer is the only framework in the Rails world that tries to pursue middle grounds. And this is my learning. Introduce a higher-level architecture than Rails does, but keep it low. And simple.

And the cool thing is: Ruby allows that!

If I decide that it's better to manually validate input of an operation, I can do that instead of using the operation's contract. If my view is simple, I don't need a cell but use a standard Rails view. If view and representer code are identical, I don't need to introduce a twin. Don't confuse this with "Omakase", though! Trailblazer has clear rules and patterns for all kinds of problems in Rails, but it offers you a high degree of freedom at the same time.

Dynamic languages require discipline and Trailblazer gently enforces this by offering a few more abstraction layers without taking away Rails' awesomeness³.

³Yessss, I've always wanted to end a book chapter with this expression!

The Trailblazer Architectural Style

Trailblazer is not only a cool name and an architectural style for building your applications, it is also a gem you can install into your application to help you to implement that style!

Unfortunately, it needs a little bit more than just running `bundle install` to take advantage of Trailblazer.

Of course, architectural styles can't be enforced solely with a Ruby gem. You as the programmer have to adopt patterns in your code from this style and use them throughout your system whereever you feel more encapsulation would be beneficial. Luckily, Trailblazer aims to make this very easy, comprising just a few conventions and classes implementing those patterns, along with this fine book to guide you through the process.

Oh, and did I mention that you are welcome to email me at anytime? Great.

Why Trailblazer?

The real question is: why not Rails?

Actually, that's a trick question because Trailblazer sits on top of Rails. You're free to use as much *Rails Way* as you want. So, why do we need Trailblazer? Isn't it a backward step to introduce more technical complexity into this amazing framework?

The answer is: No. Rails needs more abstraction layers.

Over the past years, for whatever odd reasons I became a “refactoring expert”. A “refactoring expert”. At least, that's what people think I am. I disagree. “Refactoring expert”. Say it aloud and try not to laugh.

Companies would hire me to “improve” their Rails apps on an architectural level. That means that I wasn't supposed to write new features or fix bugs but to re-structure their legacy code into a manageable, extendable architecture. And I loved doing this - and still do!

To make a long story short: in every app, and it didn't matter whether this project was in Berlin or in Munich or in Cape Town or in Sydney, in every project I found the same problems. Problems that get shipped with Rails itself, it seems.

The monolithic design of Rails basically led every application I've worked on into a cryptic code hell. Massive models, controllers with 7 levels of indentation for conditionals. Callbacks, observers and filters getting randomly triggered and changing application state where you don't want it.

Views and helpers lacking any kind of structure, partials with a ridiculous amount of ifs to make them “reusable”. Tests that basically test if the mocked mocks are mocked properly. If you touch a

mailer, you probably break a model in a completely unrelated area, if you ran a migration customers would suddenly get unsuspected emails about their account confirmation, and so on.

Now, please don't tell me your applications are clean and awesome, and everything I just said is wrong. I believe you are awesome and your apps do kick ass (I really do!), because you know what you're doing. However, many Rails developers might not have that level of expertise to judge their design, yet, they need structure and architectural guidance. That is the whole point of a framework.

When it comes to architecture, Rails has its famous "MV and C", and that's it. Oh, and, sorry, "concerns".

Vanilla Rails doesn't give you any kind of high-leveled structure.

The question I hear most from developers is "Where do I put this kind of logic?". Eventually, I figured that this is clearly a design flaw in Rails itself and I stopped blaming the other poor developers for all those thousands of emails that had just been sent to all the customers, all because I ran a migration updating some models.

And that brings me to the next problem: third-party gems hook into the same low-level mechanisms as you do. Filters and callbacks are magically added. Simple workflows, like changing a model's attributes, suddenly develop a bizarre life of their own. While this was all made with simplicity for the programmer in mind, the side effects of those hidden semantics can be devastating and ruin your day.

Some charismatic leaders in Rails core dislike encapsulation. The *Fear Of The Class*, the freedom of a dynamically typed programming language and the strong will to make it different to Java and its "over-abstraction" has led a generation of developers to unlearn what object-orientation really is about - and neglect this ingenious concept.

I like structure and reasonable encapsulation. And that's what Trailblazer gives you.

A Non Intrusive Framework

It is a design goal to allow you to step-wise introduce the Trailblazer style into your existing application without having to rewrite the entire thing from scratch.

Play with Trailblazer when implementing the next new requirement in your project. Or give it a go one afternoon and refactor a piece of existing legacy code into a concept. Extract business code and validations into an operation, write tests and watch yourself deleting code from models and controllers. It feels great.

Trailblazer was created while reworking Rails code and does absolutely not require a green field with grazing unicorns and rainbows. It is designed to help you restructuring existing monolithic apps that are out of control.

While Trailblazer was developed mostly in Rails applications, and that is the main focus of this book, the Trailblazer style can be applied to any system design. Its concepts are easily transferable to other

languages and frameworks and absolutely not bound to Ruby or Rails. I love Ruby, and the Rails community, that's why I created Trailblazer in this beautiful environment.

Trailblazer's pattern implementations found in the gem itself have zero or very little coupling to Rails. I know of many projects where Trailblazer is used in other frameworks like Roda, Sinatra or Grape and make more people happy. Oh, excuse me, "Sinatra is not a framework"⁴.

This loose cohesion makes Trailblazer a, what I call, "non intrusive framework". As stated in the README it does not missionize you. You decide where and how much Trailblazer you want and where the Rails Way is sufficient. They don't interfere with each other.

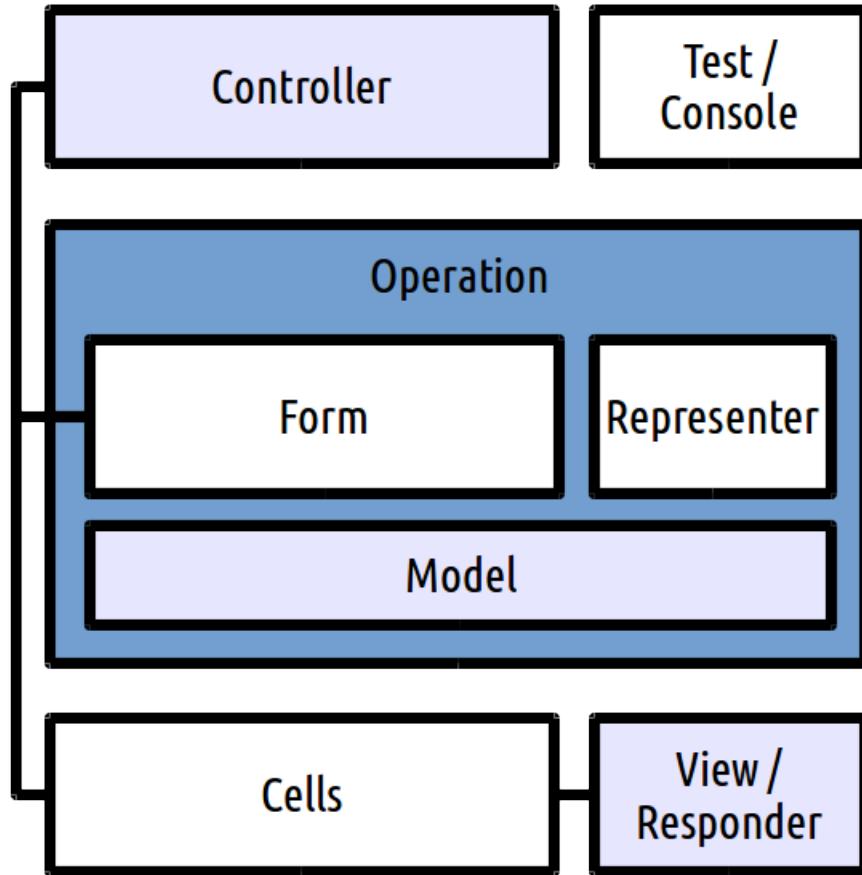
You can also pick which components from Trailblazer you find helpful. Trailblazer is a collection of gems that implement different patterns. It is up to you to judge and remove layers you don't like. The different gems integrate with each other. For example, operations use form objects per default and form objects use representers. That doesn't mean that you *have* to use all of those gems and patterns, though. Freedom.

⁴Konstantin Haase's famous last words...



Trailblazer In A Nutshell

I would like to introduce the architecture and technical aspects of Trailblazer with this incredible diagram I made for you in my sleepless nights.



The Trailblazer stack.

As you can see Rails' original components are still there, along with new abstraction layers. Here's a brief overview, then we're going to discuss the layers with a bit more detail.

The sleepless nights was a lie. My sleep is excellent.

The diagram is to be understood from top to bottom, where the top represents the incoming request endpoint and the bottom compiles the response.

1. Controllers become lean HTTP endpoints. They handle authentication (if not handled elsewhere), differentiate between request formats like HTML or JSON and then instantly delegate to the respective operation. No processing logic is to be found in the controller.
2. An operation contains all the business logic.
3. Every operation validates its input using a form object.

4. Models are lean persistence endpoints. No logic but configuration is allowed in model classes.
5. Operations and forms can change and persist models.
6. Representers can be used to parse incoming documents and to render API representations. Since form objects internally use representers they can be used for that directly without having to specify a representer.
7. Controllers can render views or delegate to Cells (a.k.a. View Models) and provide a better abstraction for rendering complex UIs.
8. Controllers might also use responders to render responses. Cells and operations can be passed directly into a responder and completely remove the rendering logic from controllers.
9. Operations replace test factories, and can be used in scripts and console, too.

Now let's check how this all works together.

Logicless Models

The thing I always mention first about Trailblazer is my favorite one.

Your data models become stupid, empty classes that solely contain persistence configuration.

Experiences from the past decade have taught us that it is a not-so-good idea to push all kinds of business logic into your model layer. With relational mappers like ActiveRecord, this often ends up with huge “model” classes that contain hundreds of lines of code and ugly conditional configurations to re-use the “model” in different contexts.

A model in Trailblazer typically looks like this.

```

1 class Rating < ActiveRecord::Base
2   belongs_to :thing
3   belongs_to :author, through: :user
4 end

```

Beautiful, isn't it?

It's got something innocent, a model without business logic, no callbacks and no validations. “*Where are the validations and all that?*” you might wonder now, a drip of sweat running down your panicking face. Relax, it's all still there, but not in the model anymore.

In Trailblazer, the model class becomes the place for persistence, only, making the model a low-level *data object* with a very simple set of functions: Retrieving and writing data to a database.

Yes, I know the term [ActiveRecord](#)⁵ originally does include “domain logic”.

⁵<http://www.martinfowler.com/eaaCatalog/activeRecord.html>

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

I doubt that the author's intention was to legitimise the creation of monstrous super objects that basically do everything, though. Martin's examples limit the "domain logic" to query methods.

And this is exactly how models are handled in Trailblazer. You can use query methods and scopes to retrieve objects. You can use models to write or update rows. And nothing more. This is the lesson learned from Rails and its monolithic "fat model" approach.

While this might seem confusing it has proven to be an extremely powerful and clean approach - and I've spoken to quite a few people who have a similar design without Trailblazer. The logicless model is nothing new.

What I love about it is the fact that I still use all of the ORM's awesomeness: I admire how ActiveRecord takes away the pain of SQL, joining tables, escaping queries, and so on. That means I use finder methods, where, scopes and what not to deal with the database - I just don't put any specific business logic into my models.

Of course, you are not limited to ActiveRecord but may use any mapper library you fancy, whether that is DataMapper, Mongoid or NoBrainer, the framework does not care.

The business logic for both reading and writing along with validations is abstracted into new layers: Operation, Form and Twin are patterns that Trailblazer brings to your architecture.

Let me now briefly outline those "patterns".

Concepts and File Structure

Rails organizes the implementation of your domain into models, controllers and views.

For example, to create a comment via a web page, this will usually be triggered in a controller. The controller then invokes one or several methods on models. After processing, a view is rendered.



Even though this code belongs into the same logical group - "*Create a comment!*" - the actual code files are spread into different directories and separated by technology.

I found it hard sometimes to navigate in Rails apps, you keep flicking through a million directories and it makes me cry when I decide to rename a model as I also have to rename at least one more directory and several files. Extracting or removing a model and its friends is the same story - it feels wrong.

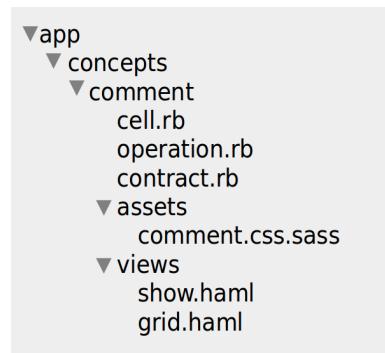
In Trailblazer, you structure your app by domain into real components. I call them *concepts*. You implement a concept for every entity in your high-level domain. That means you have comment, profile, thing, feed, or a dashboard concept.

Those concepts sit in `app/concepts` in their own directory. All classes and views belonging to that concept, forms, operations, and so on, are in that very directory.

Assuming that we're both talking about the same component I can say "the index view" and "the notification cell". You will know exactly where to look.

Personally, I find this new file structure way more intuitive and it makes it easy to see the complexity of a concept with a simple glance into its directory. If there are too many files floating around in a single concept, it might be time to split it up or extract functionality to a separate concept.

Also, this emphasizes the component character of a concept. By having a group of files in one directory you can remove, reuse or temporarily disable an entire component with one click. In case you need a finer leveling, concepts can be nested into concepts, too.



High-Level Domain

Now let's get to the fun part: code!

Trailblazer's approach here is to decouple your actual application code from the underlying framework without sacrificing all the Rails goodies. By encapsulating your business, tasks like updating Rails itself, writing isolated tests or replacing entire components become less painful.

I'll start from the top.

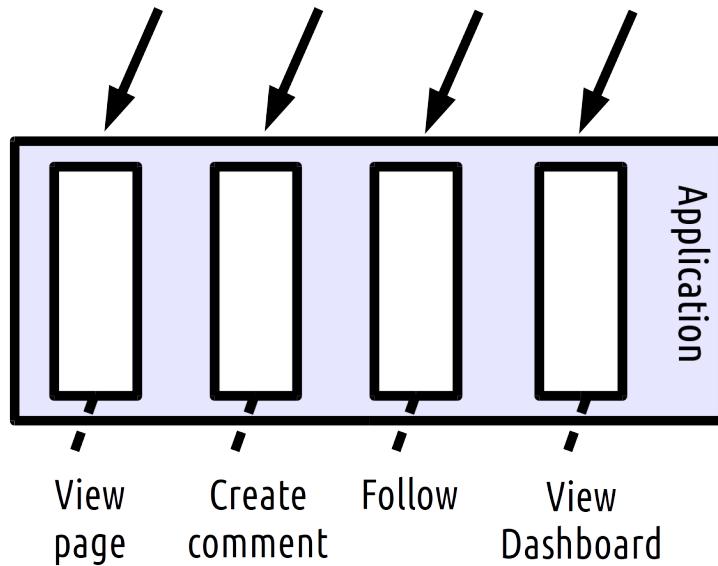
Using a web application means you send requests, that queries or changes application state, then you get a response. The browser hides that from you, but behind the scenes this is exactly what happens. I want you to think of every *function* of a web application as a request-response cycle. It's not relevant if that request is triggered by clicking a "Create" button, or if it comes from an AJAX request or some sophisticated JavaScript logic.

Fact is, you operate your application by triggering requests.

As an example, this could be the following typical session.

- Visit the Trailblazer page and read the summary.
- Write a comment using the page's form and submit it.
- Click the "Follow Trailblazer" button as you're interested in the further discussion.
- Go to your dashboard and see new comments for "Trailblazer".

These are four *functions* of the application, and they are all triggered with a separate request.



The steps of our high-level domain.

It doesn't matter what happens behind the scenes - those four functions somehow need to be implemented on the server-side. The "functions" are what I call the *High-level Domain* in Trailblazer.

In Domain-Driven Design (DDD), this is called a *Use Case*. In CQRS, this kind of "function" is called *Command*. This is your public API you're presenting to the user of your application.

Often, very often, high-level domain functions can be invoked via the web UI, via a document-based HTTP API, e.g. by exposing JSON-consuming and rendering endpoints, and sometimes, very rarely in Rails, you can trigger the exact same function by invoking a method on a model.

While Rails does provide a high-level domain in the form of controller endpoints, the implementation thereof is fuzzy. Is it implemented in the controller? Or the model? Or a service object and the controller?

You might think that code to create a comment all sits in one place, e.g. the model, but this is wrong. Code that logically belongs together is partly located in the controller, partly in the model. The high-level domain is not encapsulated in an atomic object, but distributed over several layers.

For example, the following snippet is code you will find in many Rails controllers.

```

1 def create
2   @comment = Comment.new { |c| c.author = current_profile }
3   # ... more stuff
4 end

```

Or, another misleading example for implementing a high-level domain function. This is actually from Rails' documentation.

```

1 def create
2   @project = Project.find(params[:project_id])
3   @task = @project.tasks.build(params[:task])
4 end

```

In both snippets, the controller clearly contains business logic. Even if that's just assigning the current user, or invoking a builder on a model - this is code that is part of your domain, and not related to HTTP in any way.

It's impossible to replicate the above behavior somewhere else without replicating the controller code itself.

Keep in mind that once we hit the model further logic is triggered as in validations and callbacks. Speaking of validations, this is when controllers really start to get messy.

```

1 def create
2   # ...
3   if @project.valid? and @task.valid?
4     redirect_to projects_path
5   end
6 end

```

Again, the controller gets stuffed with knowledge about model internals. Why is a HTTP endpoint aware of projects and tasks, their relationship to each other and their validity constraints? There is no way to re-use this function without duplication.

Going through the controller you will find more places where business code is implemented - in the wrong place. Other examples are the usage of `strong_parameters` where the controller suddenly knows which attributes go to which model, and so on. Or, my favorite, when `before_filters` contain additional model validation logic that should clearly sit somewhere else.

Operation

In Trailblazer, the implementation of a high-level function goes into an *Operation*. Surprisingly, this is a class!

Maybe it's easiest to think of an operation as “everything that happens in a request after the HTTP overhead is sorted”. Or in other words: the controller only knows about the HTTP layer. Business is immediately dispatched to the operation.

This underlines how each operation embraces a complete action (or *function*) of your public domain.

Without even understanding internals of this new concept have a look at how controllers typically end up in Trailblazer.

```

1 class CommentsController < ApplicationController
2   def new
3     present Comment::Create
4   end
5
6   def create
7     run Comment::Create
8   end
9   # ...
10 end

```

No business logic in controllers and an immediate dispatch to the target operation are fundamental concepts of Trailblazer. We will soon learn that the messy code is not just put away into another module but restructured into different layers of Trailblazer.

An operation class exposes a single public method, per default. I know, you might think this is clumsy and backward, but throughout this book we're gonna work a lot with operations and I think I will manage to demonstrate why I love this functional feature and the simplicity of an operation.

```

1 class Comment < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     def process(params)
4       # ...
5     end
6   end
7 end

```

Operations usually get implemented in their concept's namespace. Here, the create code gets encapsulated into a separate class and namespaced into `Comment::Create`, making it pretty obvious what is the classes' responsibility.

Note that putting classes into classes is simply namespacing and doesn't involve inheritance or any sort of coupling.

An operation always has to implement one method: `#process`. This is where the implementation of that use case sits.

Usually, you'd invoke an operation using the *call style*.

```
1 Comment::Create.(comment: { body: "Awesome!" })
```

And here we are, this is our high-level *entry point* for creating a comment!

That being said, the controller really boils down to dispatching to “its” operation.

```

1 def create
2   Comment::Create.(params)
3 end

```

Now, wait! That's only half of it. An operation can also be used from the console or as a test factories.

```

1 let(:comment) { Comment::Create.(comment: { body: "Testing rules!" }) }

```

An operation provides a single entry point for domain functions: There's only one way to create a comment and that is the operation - unless you provide additional operations to do so.

This is a good thing. Conventional factories in tests create redundant code that never produces the exact same application state as in production - a source of many bugs. While the tests might run fine production crashes because production doesn't use factories. In Trailblazer factories get superseded by using operations.

Operations can easily be subclassed and extended, for example, to parse or “understand” a JSON string instead of a hash.

```

1 Comment::Create::JSON.(request.body)

```

Without going too much into the details: An operation doesn't just blindly traverse through a deserialized JSON hash. It knows about the document structure and semantics because you configured it. We'll come to this very shortly.

An Operation is a Service Object

Those specific concepts and behavior of operations are pretty unique to Trailblazer, I haven't seen this anywhere else. However, the idea of encapsulating code from controller and model into a service layer is old.

An operation is pretty similar to a “service object”.

The Ruby community is becoming obsessed with service objects - a result of Rails' lack of a defined layer for domain logic. People now start to extract code into separate “service objects”, whenever you ask someone where to put this or that code the answer is gonna be “*Use a service object!*”.

Especially for unexperienced developers this advise is not helpful at all. It might end up with a set of different “service objects” implementations in one app, varying interfaces and concepts being applied. This defeats the purpose of Rails' conventions, which are said to make it easy for fresh developers to understand new projects.

Instead of leaving it up to the programmer how to design their “service object”, what interface to expose, how to structure the services, Trailblazer's Operation gives you a well-defined abstraction layer for all kinds of business logic. Its implementation is incredibly simple but it offers a bunch of neat features that implement a lot of best practices I've found handy in the last years.

Business Logic

Once you map your high-level endpoint to an operation, all kinds of business logic will go into this class, or into nested operations.

This is not limited to classy CRUD code. An operation can implement all kinds of public API, such as following a user, cropping an avatar image, retrieving a list of comments for a search term, and so on.

Even though this happens explicitly in a separate class per function, you don't have to code everything yourself. Trailblazer comes with many helpful modules to extend operations so you don't need to do all the work. For example, creating or finding models is implemented in the `Model` module. I am not going to use any of those helpers for now to give you a clear image of what is the point of Operation. We'll come to those features later in the book.

You may run any logic you want in your `#process` method - Trailblazer doesn't impose any limitations. It guides you, but you're still free.

```

1 class Comment < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     def process(params)
4       Comment.create(params)
5     end
6   end
7 end

```

Here, it becomes obvious that each operation maps to one CRUD action (and more). Well, admittedly just calling the `create` method in a clumsy operation doesn't really convince me to model my business code with this new structure. The real power of operations comes with the validation, processing and post-processing of the data, and Trailblazer structures this workflow in a pipelined fashion.

High-Level Architecture

Again, everything that happens between the interception of the request by the endpoint, and the delivery of the rendered result, in Trailblazer is business logic and embraced by one operation.

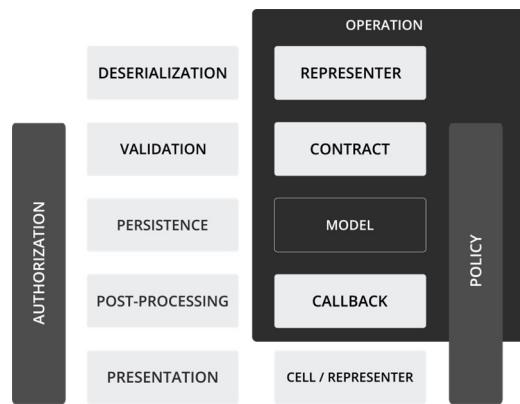
The operation here acts as an *orchestrator* dispatching between the different layers of Trailblazer. The layers are all separate objects that have no idea about the context they're being used in, or what an operation is.

Often, the operation is criticized as a *God Object*, but those people confuse *implementation* and *orchestration*. Somewhere in your code, regardless of whether this is a set of Elixir functions or a super object-oriented Ruby system, somewhere you have to dispatch the actual steps to process the request and run your business logic.

And in Trailblazer, this is the operation. The operation doesn't know about implementation-specifics, only where and how to invoke the responsible objects.

Since we all do appreciate diagrams, here's a typical web workflow schematized.

In Trailblazer, we chunk the processing of a request into deserialization, validation, persistence, processing and post-processing and presentation. Each step maps to a dedicated layer in the operation and is implemented using one or more self-contained objects.



1. **Deserialization** means transforming the incoming request data into an object graph. This happens with a representer.
2. **Validation** we call the process of verifying this object graph is in valid state. Validation's done by the form object. In Trailblazer, we often say *contract* instead of *form*.
3. **Persistence** will write the application state from the object graph to the database. Usually, this happens by syncing the graph to models.
4. **Post-processing** is what happens before and after data gets persisted. In operations, you have callback objects that can be invoked to run additional logic.
5. **Presentation** is not directly the operation's responsibility. However, operations can assist rendering a JSON document using a representer, or by providing data to view models.
6. **Authorization** realistically is a task orthogonal to this flow stack and can happen at any point leveraging policy objects.

Structuring and implementing the typical requirements of a web request in Trailblazer is called *High-level Architecture* and is the missing piece of frameworks like Rails. Without a high-level architecture, developers often complain about missing structure in their code and problems that arise from lack of encapsulation.

Validations

When writing Trailblazer I didn't really know about the role of Operation and if it's worth introducing. This was until I combined it with a form object. Guarding your business logic with a validation simply makes sense and feels very natural.

Another free feature is that the contract can be used for deserializing (and serializing) incoming documents, e.g. for a JSON endpoint. Let me come to this later, but keep in mind that a contract is a *schema* of the data structure.

An operations lets you define a form object, or *contract*. Internally, this simply creates a Reform⁶ form class. If you've used this gem before, you will recognize the API.

```

1 class Create < Trailblazer::Operation
2   contract do
3     property :body,    validates: {presence: true}
4     property :author, validates: {association: true}
5   end
6
7   def process(params)
8   # ...
9 end
```

This is pretty straight-forward, isn't it? In the contract block you can define properties of your form and their validations. If your form is more complex and needs to validate, create or update compositions of models you might define nested properties and populators.

Likewise, since Reform uses ActiveRecord::Validations, you can use all kinds of standard validations like length or inclusion. Specific logic e.g. to validate multiple parameters can be achieved using the good ol' `::validate`.

Ok. Fine. We got a class that is supposed to contain our business logic. This "operation" has a form object that defines the incoming document structure and specifies validations to assert a valid application state. Now, how does this all play together?

As soon as you define a contract for an operation you can use it for deserializing the incoming parameters and validate the object graph that was created. This sounds crazy but is extremely simple. Check out the `#process` method now.

```

1 class Create < Trailblazer::Operation
2   contract do
3     # ...
4   end
5
6   def process(params)
7     @model = Comment.new
8
9     validate(params[:comment]) do |f|
10      # process the data here
11    end
12  end
13 end
```

⁶<https://github.com/apotonick/reform>

The operation gives you the `#validate` method to take advantage of your contract. This is a totally optional feature: you don't have to use a contract and the validation if you don't need it.

Again, the *contract* is simply a Reform object. Bear with me, I'm gonna talk about Reform in detail throughout this book. For now, let's focus on what `#validate` does internally.

1. The operation instantiates a contract and passes its `@model` to it. In our case that's the `Comment` model. We will soon learn why the form wants to wrap a model.
2. It then instructs the freshly created form to validate the params hash.

It is absolutely mission-critical to understand that this step does *not* touch the model at all. Validation will only operate on the form instance. And this is different to Rails: Validations in ActiveRecord happen on the model itself. Reform cleanly separates that and embraces the entire validation workflow in the form object.

3. If the validation was successful without errors, the block passed to `#validate` is run. This is where you put your business logic for processing the form.
4. If the input was invalid, the block is *not* run.

And that's basically it.

I find it important to mention how `strong_parameters` becomes obsolete using operations and forms. The `#validate` method knows which parameters of the hash go into the form and what is to be ignored.

How does it know that? Well, you configured the form's fields and validations in the `::contract` block! The contract just needs to pick the keys it wants from params. Explicitly defining structures instead of guessing is a cornerstone of this framework - and will help us a lot in our quest to a cleaner architecture.

Processing Data

Let's see how a real `#process` method looks like in order to discuss the data processing steps.

Again, this is a bare-bones operation without using any of Operation's built-in functions to ease your life. Even though it might look un-appealing to you, I do this on purpose. This section is not a show-off of how much less code or how much more readable structure this pattern gives you - I want you to understand what is going on on the fundamental level. Then we can move on to abstractions.

```

1 def process(params)
2   @model = Comment.new
3
4   validate(params[:comment]) do |f|
5     f.save
6
7     notify!
8     log!
9   end
10 end

```

In case of a successful validation, what we're all really hoping for, the logic in the block gets executed. In my simple example, two things are happening.

The `#validate` method yields the contract instance to the block. This is helpful to access the validated data. As we're gonna discover in the next chapters, the Reform contract comes with a handful of public API methods. One of them is `#save`.

All this invocation does is push changed values from the form to the `Comment` model (this is called *syncing* in Reform). Afterwards it calls `#save` on the model itself, and thereby persists the validated changes from the form. This is what usually happens in `Comment.create(..)` or `update_attributes` call when using models directly in controllers.

While letting the form take care of persisting the data here it is fully up to you how you work with the database layer and the models. In later chapters we're gonna go through other ways how to store models in the database, by-passing the form object's save semantics.

After persisting the model, I run arbitrary code which is similar to `:after_create` hooks in ActiveRecord. I just outlined this by calling `notify!` and `log!` so it becomes obvious that you can do whatever you want. The operation provides you access to the model and the contract.

This is where you'd also expire page caches, send out emails or invoke further business logic. This doesn't have to be hand-coded in this very operation but can be delegated to other classes, nested operations, or callback objects as we're gonna discover in chapter 8.

Operations provide a clean way to group callbacks. Instead of adding ActiveRecord callbacks with conditionals you explicitly invoke the hooks in your operation class. In Rails, this is often done with ugly `:unless` or `if:` lambdas and the like. You never know whether or not a callback gets triggered when saving an object. This creates fear. In Trailblazer, you ideally expose a new operation if you decide you need the same behavior without any "callbacks". Trailblazer offers some nice techniques to derive operations with inheritance.

This code is very explicit. Some might call it verbose, because "in Rails, this is one line". Well, this might be true in some rare cases. However, we haven't looked into Operation's beautiful add-ons, yet, that can abstract most of this code. Even in Rails you still need to define validations (probably with conditionals) and callbacks. These callbacks are then later gonna shoot you in the foot.

Have you ever tried to change what happens *in* your “Rails one-liner”? Exactly, mission impossible. It requires a shocking amount of patience to step through ActiveRecord’s magic of deserialisation, validation, callbacks and persistence logic all happening in one class.

The goal of the *Operation - Contract - Model* design is to separate concerns. This does not only make it easier to follow the flow of your logic it also maximizes reusability. You can reuse massive parts of the operation for other domain actions like updating a comment, to process differing formats like JSON and also reuse the operation in other contexts, for instance in an admin backend.

Inheritance Over Configuration

In a perfect CRUD world, create and update semantics are identical. They consume the exact same set of attributes.

There’s no borders - users and admins share the same rights and privileges, you simply re-use forms and validations for both the frontend and the admin interface.

You also don’t need configuration for different request formats. You simply pass the `params` hash into your model’s `create` or `update` method without even having to worry about who or where this JSON request or a form submission was magically deserialized into a hash.

Your HTTP API works identical to your forms, the incoming JSON document and the serialized form have the same structure, you can even pass a self-made hash to the `create` method, everything works automatically, everything is great.

In your dreams.

What really happens is: Your update action requires a sub-set of the create parameters. Your JSON API consumes completely different documents that do not have much in common with your form submission hash, and the form needs different fields for processing than the manual hash you use in the console.

A form to create a comment in the user’s UI has a set of fields and validations that has almost nothing in common with the way an admin can post and edit comments. You realize how many different contexts you have for one model and get frustrated.

Trailblazer was designed with differing endpoints in mind. Where the “Rails Way” offers a baffling amount of tools to solve this problem in the controller with `ifs` and `else` and `strong_parameters` and `before_filters` and `responders` and `variants` and `format deciders` and additional logic and configuration in the models, Trailblazer copes with this using a completely different approach.

Per context you can maintain a sub-class of the original operation. A context might be a document request format as JSON, a follow-up action like `Update` or a differing environment, for example the

admin backend. Ideally, in the above mentioned dream-world, you'd handle this with one and the same operation class in every context.

However, when things get dirty - and software engineering is dirty - when different formats need different flows, structures and semantics, you have a sub-class to deal with that, without interfering with the other format operations and without any ifs and elses in your control flow.

The explicit nature of Trailblazer might seem clumsy at first glance. Nevertheless, when dealing with different contexts, you will feel the power and ease of real object-orientation.

Subclassed operations will inherit behavior, the contract and representer.

```
1 class Update < Create
2   action :update
3 end
```

Often, it is sufficient to simply subclass. I prefer a two- or three-liner over implicit semantics, however, note that this can happen automatically, so you don't have to write a single line of code.

After inheriting you're free to change, override or rewrite contracts, business logic and representers. Besides Ruby's built-in object-orientation, Trailblazer offers you some helpful ways to tweak operations and contracts.

```
1 class Update < Create
2   action :update
3
4   contract do
5     property :author, readonly: true # make author read-only for update.
6   end
7 end
```

An update is never identical to a create. And Trailblazer makes it as simple as possible to map these requirements to code - using clean polymorphic classes without conditionals.

Especially when extending operations to handle JSON inheritance becomes a powerful tool you're gonna love.

```

1 class Create
2   class JSON < self
3     include Representer
4
5   representer do
6     property :author do
7       property :name, as: :fullName
8     end
9   end
10 end
11 end

```

The explicit code makes it straight-forward to understand what behaviour is changed in format-specific operations and contracts, here for a JSON request.

And this is because Trailblazer is designed to handle polymorphic domain logic, this is not just another “quick” feature you push into the existing codebase using if/else and the like, as I have found it in many vanilla Rails projects.

Builders

Instantiating sub-operations, forms and cells according to the environment is built into Trailblazer - you define when to create which object and the framework will take care of the rest. This is extremely helpful to keep your controllers clean while still allowing access to all your subclasses explicitly, for example when working from the command line.

The trick with builders is that the caller doesn’t know about the polymorphic semantics of the operation - polymorphism in OOP was introduced to hide internals like that.

```

1 def create
2   Comment::Create.(params)
3 end

```

Here, the controller simply invokes the top operation. Now what if we need to distinguish between creating a normal comment and a moderated comment? You’d implement that with an operation `Comment::Create` and then inherit and extend in `Comment::Create::Moderated`. The latter would send out additional notifications for moderators, and so on.

I don’t want my controller to know about all that. Where the controller dispatches to `Create` the operation class itself takes control of building its concrete instance.

How does the operation know which subclass to instantiate? You configure it using the builder API.

```

1 class Create < Trailblazer::Operation
2   builds ->(params) do
3     Moderated if params[:current_user].needs_moderation?
4   end
5
6   class Moderated < self
7     # ...
8   end
9 end

```

This will create different instances for different environments: The operation will run the builder block to figure out which concrete class is appropriate to handle the environment. Note that the params hash needs to contain all necessary data to figure that out! This is the only requirement to the caller - which usually is the controller.

```

1 Comment::Create.(current_user: admin) #=> Comment::Create
2 Comment::Create.(current_user: nick)  #=> Comment::Create::Moderated

```

Keeping that knowledge in the operation and out of the caller environment like the controller allows to use your domain virtually everywhere without having to replicate logic.

The builder feature is a fundamental concept found in all layers of Trailblazer. I started experimenting with it in Cells where it allows to render polymorphic widgets and collections without magic - and got positively surprised by the level of acceptance in the community.

Authorization And Policy

Another critical requirement that Rails completely leaves up to the developer is that every application needs authorization for different actions. Different users have different permissions. I don't want everyone to screw with my profile or to post comments in my name. There's no dedicated place for this in Rails, every application has a slightly different approach.

This logic usually ends up in as a mix of `before_filters` in controllers and conditionals in model methods. It's not only hard to follow what is going on, also does it create redundant code. Neither the HTTP layer nor my database is the place to put authorization into. The lack of a permission system is a grave design flaw in Rails resulting from the core team's resistance to introduce new abstraction layers.

In fact, authorizing actions for differing environments is an essential part of your system requirements. This cannot just be handled in controller filters. Authorization needs knowledge about the domain and access to the environment. In other words: it needs to be integrated into your business.

Trailblazer offers you to sort authorization as part of an operation, taking this logic back to where it should be: your domain layer.

The simplest place for permission handling is in builders. The benefit here is that the created class doesn't need to know anything about rules and can perform its task without looking back. Permissions are computed in the builder and then the permission is granted by creating subclassed operations with differing behavior.

```

1  class Create < Trailblazer::Operation
2    builds ->(params) do
3      user = params[:current_user]
4
5      return Create if user.admin?
6      return Moderated if user.can_post?
7      deny!
8    end
9
10   # ...
11 end

```

Right, I use ifs and else here - this is builder code and all decisions are made in one place. In fact, builders are the only place where deciders of that level should be tolerated.

Just like the operation, the preceding builder has access to the params passed into the operation call. It's up to you what authorization framework you use here. Use cancan, rolify, pundit, or write your own, everything is allowed (or denied). While we build our app in this book, we will use pundit-style authorization.

By building different sub-operations we implicitly give out permissions. The sub-operation is decoupled from the authorization process which makes it simply to bypass the permissions: by calling the sub-operation directly the builders are ignored.

```
1 Comment::Create::Moderate.(current_user: nick)
```

This is extremely helpful for console work, scripts or tests.

Another way to let an operation do the permission work is using a *policy* block in your operation. As you might have guessed already a policy block will skip the operation if it evaluates to false. Of course, builders and policies can be combined if you need that fine level of permissions.

```

1 class Create < Trailblazer::Operation
2   policy do |params|
3     Permit.can?(:update, model)
4   end
5
6   # ...
7 end

```

The policy block is run after the setup of the operation but before the processing, allowing you to access the model and other objects from setup.

I like to stress how you may use permission gems as much as you please. Trailblazer does not try to replace all the excellent gems but provides a structural location to call them.

Authentication

Signing in users and distributing cookies to authenticate them in subsequent request is not handled in Trailblazer itself. This task is mostly coupled to HTTP and thus happening in controllers.

You're free to use [Devise](#)⁷, however, don't hit me up if things suddenly go wrong and you don't know where to look. Devise is incredibly hard-wired into Rails itself and I find it surprising that it is not officially part of core like `strong_parameters` or `turbolinks`.

The developers of this gem absolutely deserve respect for their work - Devise covers an astonishing range of features and makes signup, signin, password maintenance and confirmation mails a no-brainer as long as you don't try to change anything.

The way Devise hooks into models, controllers, routes and views makes it a bit unattractive, in my opinion. I would prefer if devise had a decoupled core engine with additional Rails bindings. When in recent devise versions it turned out that the only way to get the confirmation token for the signup mail is via a global view instance variable `@token` I knew it's time to look for an alternative.

And this is why we will use an extended version of [Tyrant](#)⁸ in this book to authenticate users and cookie management.

```

1 class Session::SignIn < Trailblazer::Operation
2   def process(params)
3     Tyrant::SignIn.(params)
4
5     mark_online_activity!
6   end

```

⁷<https://github.com/plataformatec/devise>

⁸<https://github.com/apotonick/tyrant>

Tyrant itself is built using Trailblazer operations, and in chapter 9 we will learn how to use this decoupled gem for authentication.

So far we covered how an operation replaces most of the controller logic. It validates incoming data using a form object and then processes the sane data. Now, I want to show you how an operation can be used on the other side, the presentation, e.g. to render its form in a view.

Working With API Documents

In most Rails applications, controller actions act as endpoints for HTTP APIs, too. Usually they render and consume documents of a certain format, e.g. JSON or XML. Rails comes with several rendering engines to compile JSON formats.

To name some, there's the built-in and loathsome `Object#to_json` method that creates generic representations, `jbuilder`⁹, `rabl`¹⁰ or `ActiveModel::Serializer`¹¹.

Those are all excellent gems that do a great job rendering documents. However, they are all one-way template engines, nothing more. What is completely ignored in the vanilla stack is how to handle POST, PUT and PATCH requests that actually change application state by parsing and processing documents.

Unfortunately, there is no consuming layer in Rails. Parsing the document happens automatically somewhere in a Rack middleware. Unless you're using `accepts_nested_attributes` (I don't hope so) you're completely left alone to populate objects from the parsed hash.

You still have to process incoming documents yourself and populate objects manually by crawling through deeply-nested hashes and many `ifs`. This is not a pleasant task, and in my opinion, it's wrong, as you distribute your document syntax and semantics over the entire stack. If a key in the document changes, you need to fix it in the template or the serializer *and* in your parsing code.

Again, Trailblazer handles this with a different approach. I hope you're not getting tired of this.

Representers

Rendering and parsing documents in Trailblazer happens with the help of representers - another pattern you're gonna use a lot in this book.

Representers in Trailblazer come from the `Representable`¹² gem.

Maybe it will help to actually show you how representers look like and work? I promise it won't take long as they're really simple.

⁹<https://github.com/rails/jbuilder>

¹⁰<https://github.com/nesquena/rabl>

¹¹https://github.com/rails-api/active_model_serializers

¹²<https://github.com/apotonick/representable>

```

1 class CommentRepresenter < Representable::Decorator
2   include JSON
3
4   property :body
5   property :author do
6     property :email
7   end

```

That looks pretty much like a form without validations, right? And the truth is: a form is nothing more but a representer with some data transformation logic and validations!

You can then render documents.

```
1 CommentRepresenter.new(comment).to_json #=> "{\"body\": \"Wow!\", ...}
```

When rendering, the representer will follow the document structure you defined and compile a document that can be nested, contain collections, hypermedia, whatever you fancy. Even better, representers can also handle XML and YAML in case you want to be the first person alive exposing a YAML-Hypermedia API.

Representers would be lame if that's it. They can also do the same the other way round and parse documents back to objects.

```

1 CommentRepresenter.new(comment).from_json("{\"body\": \"Really?\", ...}")
2 comment.body #=> "Really?"

```

This will populate the `comment` instance with new attribute values from the document, deserialize nested objects, instantiate or build new objects, and so on. We'll learn everything about representers in chapters 11 and 12.

The crucial thing about representers is that they maintain any kind of knowledge about the document in one place. While rendering documents is provided with fantastic implementations, Rails underestimates the complexity of deserializing documents manually. Many developers got burned when “quickly populating” a resource model from a hash. Representers make you think in documents, objects, and their transformations - which is what APIs are all about.

API: Parsing and Rendering

Skipping the details I want to jump right into how we can use all this. Let's extend the `Comment::Create` operation to render JSON. Extending to handle new formats is implemented in a subclass, per convention.

```

1 class Comment < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     # ... all the code from earlier
4
5     class JSON < self
6       include Representer
7     end
8   end
9 end

```

This creates a subclass `Create::JSON` that is responsible for building new comments using JSON. It also inherits the contract. Now, the most generic use-case in a controller is parsing an incoming JSON document, creating a comment, and then rendering a JSON representation of the fresh comment. This could work as follows.

```

1 def create
2   respond Comment::Create
3 end

```

This minimal code will automatically instantiate the `Comment::Create::JSON` subclass for JSON requests and run it. The operation then deserializes the document using the representer from the contract, runs the business logic and then gets passed to `#respond_to`.

Now, it's back to the controller to invoke rendering and handle the HTTP response. The responder typically calls `#to_json` on the operation which was passed into it. The operation will use its representer to serialize a document, and the controller simply uses this for the response.

This keeps the controller free of any business logic. The operation in turn delegates rendering to the representer staying free of any representing code. Note that you don't change anything for the business logic. If you need to, you can override the `#process` method.

Naturally, the form's structure won't exactly match your JSON document. No big deal, an operation makes it really easy to extend or completely override the representer.

```

1 class Create < Trailblazer::Operation
2   #
3
4   class JSON < self
5     representer do
6       property :related_comments do
7         property :id
8       end
9     end
10    end
11  end

```

You can customize the operation's representer using the `::representer` block. Just like the `::contract` method internally creates a Reform form, this simply works on a `Representable::Decorator` class that is generated for you from the operation.

Using Hypermedia

Representers are designed to implement object-oriented documents for REST APIs. This implies a strong focus on embedding and consuming hypermedia in the documents. And luckily, we can use the [Roar¹³](#) gem which extends `Representable` and makes using hypermedia a pleasure for creatures great and small.

In chapter 11 and 12 we will dive into building hypermedia APIs with Roar and Trailblazer. As an appetizer here's an example how simple it is to render and parse HAL-JSON compliant documents with Ruby objects.

```

1 class JSON < self
2   representer do
3     include Roar::JSON::HAL
4
5     link :self do
6       comment_path(represented.id)
7     end
8   end
9 end

```

The rendered JSON document representing the comment will now include a `self` link following the HAL-JSON standard. Of course, the `HAL` module comes with plenty of more features to support rendering and parsing HAL documents.

If you decide that JSON-API is more suitable for you as you're maintaining an Ember.js frontend you can do so.

```

1 representer do
2   include Roar::JSON::JSONAPI
3
4   # ...
5 end

```

Roar supports HAL and JSON-API media formats out-of-the-box, including hypermedia, embedded resources (called *compound* in JSON-API) and link templates. It took a while for the community to

¹³<https://github.com/apotonick/roar>

realize that Roar is more than a template gem that renders documents. People now appreciate Roar's ability to also deserialize documents back into objects using the same definition schema.

Processing HTTP APIs using Trailblazer's presenter pattern is a very joyful experience as we will see in later chapters. Often, the integration with forms and controllers allows it to implement a full-blown API with very little code. And Representable and Roar have some nice features to make deserializing complex object graphs as painless as possible.

Rendering Views

Now after all this backend talk I want to become human again and speak about the visual user interface side of Rails. This is how Rails got big back then.

View *helpers* make it incredibly simple to put up complex forms, links or display images without having to fight with markup and HTML specifications. I find the implementation of helpers in Rails questionable but I love how form helpers make my life easier - they actually *help*.

Unfortunately, the view layer in Rails is one of the most neglected components in this framework. It's surprising, and not in a good way, that it actually takes five major Rails versions to finally decide to "refactor ActionView". Since its inception in 2005 the Rails view layer hasn't changed a single bit from an architectural perspective.

It provides parsing templates and substituting placeholders via controller instance variables and locals. Helpers are global functions (not methods) you can use to "encapsulate" behavior in your template. ActionView also allows rendering partials, again, to "encapsulate" template markup and make it reusable. The entire rendering process is run in one instance, every helper and partial can *and does* access global state.

Frankly, I can't see any difference to PHP scripts here. What I totally *can* see is how this is sufficient and efficient for simple pages. Everyone instantly grasps how to push data into the views and arrange it nicely. That's why Trailblazer still allows you to work with controller views.

Controller actions can invoke `#render` the way you know it from Rails. Earlier we learned that we can use the form object from an operation in views and present HTML forms. Operation also provides you the model and the operation instance itself if you need it.

```
1 def create
2   run Comment::Create do |op|
3     # valid operation
4     return redirect_to comment_path(@model)
5   end
6
7   render action: :new # has access to @model
8 end
```

Here it is... good old Rails view rendering. Running the operation will yield the instance to the block which is only executed for a successful validation. The `#run` call also assigns `@model` and `@operation` instance variables in case you need them in your view.

That means you don't even need to rewrite your views when replacing controller/model logic with Trailblazer's operation.

Nevertheless, controller views should be handled with care. There is a great temptation to quickly add this and that instance variable and render another partial in another partial. Soon you lose track of the dependencies between views and the controller - because there *is* no interface for views and absolutely no encapsulation.

Cells

Out of my very early frustration with the Rails view layer emerged the [Cells¹⁴](#) gem. Cells provides proper object-oriented encapsulation for views. It has moved on a long time ago and we no longer use ActionView in the 4.0 release which makes me sad and happy at the same time.

Sad because we were constantly attempting to improve ActionView with the learnings from Cells but no one in core was really interested. Happy because removing this jurassic dependency has sped up the rendering several times and minimized logic to a few dozens lines of code.

Cells let you implement parts or blocks of your UI in a separate component, like a separated mini-MVC stack.

Older versions of Cells had a pretty controller-like semantic. You'd assign instance variables in *states* (like actions in controllers) and that would make the data available in the cell views. This all still works in Cells 4.0, however, I want to quickly introduce you to the new way of writing cells called *view model*.

Rendering a cell (or, view model) ironically works via a helper. Cells can be invoked just anywhere but mostly you're gonna use them in views and controllers to replace a helper/partial-mess with a decent view component.

```

1 %h1 Latest Comment
2
3 = concept(:comment, Comment.latest)
```

The above snippet could be in a controller view or the application layout.

Here, the `concept` helper will call the cell's `#show` state. Note that I pass the latest comment model into the cell - every cell requires an object to wrap, whether that is a model, a collection or an `OpenStruct` is completely up to you.

And now guess how a cell is implemented? Right, as a class! So many classes, who's gonna clean up that mess?

¹⁴<https://github.com/apotonick/cells>

```

1 class Comment::Cell < Cell::ViewModel
2   def show
3     render
4   end
5 end

```

Per convention, the `#show` method is invoked once the cell is used. The above state does nothing more but rendering its view.

Cell views are not in the global directory. They are components so views do reside in the concept's view directory, for instance `app/concepts/comment/views/show.haml`.

At first glance, cell views look identical to ordinary views in Rails.

```

1 #recent
2   = body
3   .author
4     = author_link

```

“This view is nice and tidy. That doesn’t look like an ordinary Rails view!” you might think now, and you’re correct. In Cells, instance variables and locals are still available but proscribed. The preferred way of getting data into the view is with reader methods.

An interesting change in Cells is that the concept of “helpers” doesn’t exist anymore. Methods in the view are always called on the cell instance itself. The cell *is* the view context.

To make this view working we need to provide those two reader methods in the cell class.

```

1 class Comment::Cell < Cell::ViewModel
2   def show
3     render
4   end
5
6   private
7   def body
8     model.body
9   end
10
11  def author_link
12    link_to model.author.name, author_path(model.author)
13  end
14 end

```

Again, every method in the view is called on the cell instance which is why I added `#body` and `#author_link` to the cell. These two methods provide a solid interface to the view and will be my exclusive way to populate the view with data. As you have already noticed you got access to the comment instance via the `#model` reader that is provided by Cells.

Have you also seen that it is totally ok to use helpers in a cell? No one ever said that helpers suck. As long as they *help* they're fine. In this example I call `#link_to` to render a hyperlink while making use of a URL helper to compile the address.

Accessing attributes from the decorated model is a task so common that Cells offers you a quicker way to do this. Check out the following code and how simple the cell is now.

```

1 class Comment::Cell < Cell::ViewModel
2   def show
3     render
4   end
5
6   private
7   properties :body, :author
8
9   def author_link
10    link_to author.name, author_path(author)
11  end
12 end

```

Just as in representers and contracts, cells allow you to define properties of the wrapped model. A property in cell is automatically exposed as a reader to the view.

Cells is my oldest gem and I've been working on it for almost 10 years now. The first time I actually used it was about a half year ago. I have to say I really enjoy Cells. They allow encapsulating a certain page block without having to worry about the environment. Cells, once implemented and tested, will just work.

View models are the perfect counterpart for operations in Trailblazer. They embrace view logic that is only for the HTML user interface and provide an implementation standard for reusable widgets.

There's many other features we're gonna explore in the book. Cells allow nesting, polymorphic collections, have a clean way of caching states and offer you view inheritance. In upcoming versions Cells will have a neat mechanism to inherit and override blocks in views.

View inheritance is something that is completely ignored in Rails. This is not because view inheritance is a bad thing or overly complicated but because Rails views are all bound to controllers. No one wants to derive controllers just to inherit views.

What I'm trying to say is: Rails' view architecture is far from being sophisticated. I don't want to criticise Rails too much but every other MVC framework has a much richer view tier. I honestly don't know what is the reason for this development in Rails. And I don't care anymore because we got Cells to fix it - and you're gonna learn everything about helpers, object-oriented partials, reusable templates and polymorphic views in this book.

Twin

While most of the logic is gonna happen in operations you still need a place for presentation and decoration of models. A typical example would be a reader method `#public?` that returns a boolean stating the visibility of a single comment.

Going further, this method is then to be used in a cell for UI presentation and in an operation. We all agreed to not put any logic into the model: you're free to put this code into a decorator.

Trailblazer comes with a simple decorator pattern called *twin*. Twins are used everywhere behind your back and the concept is so useful that I made it a public conceptual pattern.

```

1 class Comment::Presentation < Disposable::Twin
2   property :body
3   property :thing
4
5   def public?
6     thing.public?
7   end
8
9   # ...
10 end

```

A twin decorator makes logic reusable and we're gonna take advantage of them in some cases when we need to share behavior between layers.

Twins can also help you re-modelling your data. This is a helpful tool when working on an existing codebase with a legacy table structure. Say you want to combine the comments table and the authors table into one object in your domain to hide the fact that this entity requires two database rows.

A twin can implement a Composition for you.

```
1 class Comment::Opinion < Disposable::Twin
2   include Composition
3
4   property :body, on: :comment
5   property :email, on: :author
6 end
```

When instantiating the twin needs both objects.

```
1 Comment::Opinion.new(comment: comment, author: comment.author)
```

The user of this twin object doesn't need to know about the internal data structure. There's a couple of more nice structuring helpers in twins that we will see later.

Testing

I honestly haven't followed the whole "*Is TDD dead?*" debate and from the little pieces I read I can tell that there's a lot of misunderstandings going on between the fighting parties.

If you want to write sustainable code and maintain good sleep quality simultaneously you have to write tests for your code. Period. Arguing about "yes" or "no" is simply ridiculous and a waste of sleeping time. And I don't think anyone meant to say "*Testing is bad!*".

When writing gems you learn to write tests. Edge-cases, bugs, implementation details, new features, and so on. Everything has to be tested properly. It is a terribly awkward moment when you release a new version and break people's code.

You also learn *how* to test. I used to test the shit out of every private class. This is redundant and blocks you when refactoring. It's not always easy to decide what to test but I prefer having more tests than necessary.

That said, testing is another fundamental concept in Trailblazer. The layered architecture, using operations for both domain and factories, and simple access to your business code makes testing actually enjoyable. It is different to Rails and we will see why in the countless tests I will make you write in the book.

Trailblazer makes you write four different levels of tests.

Integration tests assure that your endpoints (or, actions) do the right thing per request format. They are what I call *smoke tests* that test the happy path and the failing alternative.

This is really simple to identify since every controller action maps to exactly one operation, that has a valid and invalid state, only.

```

1 test "POST /comments" do
2   post "/comments", {comment: { body: "Awesome!" }}.to_json
3   assert_redirect_to comment_path(operation.model)
4 end

```

Integration tests usually cover the raw HTTP-specific details, only. I don't check the integrity of the created comment, but the wiring between controller action and operation.

Operation tests are the bread and butter of your application test suite. Since operations embrace your business, you're gonna test everything that could go wrong and right.

```

1 describe "Create" do
2   it "is valid" do
3     op = Comment::Create.(comment: {body: "Awesome!"})
4
5     op.model.body.must_equal "Awesome!"
6     op.model.persisted?.must_equal true
7     op.message.must_equal "Comment for Traiblazer was created!"
8   end
9 end

```

The clumsy class of an operation suddenly becomes extremely simple to use and test. I can tell you testing operations is actually fun.

And what is incredibly convincing is the fact that we will use operations as test factories.

```

1 describe "Respond" do
2   it "[valid]" do
3     comment = Comment::Create.(comment: {body: "Awesome!"}).model
4
5     op = Comment::Respond.(id: comment.id, comment: { })
6   end
7 end

```

No more leaky factories that consistently result in a different application state. Instead use “production code” in your tests. I cannot repeat how much simpler and better my tests became with the *operation-as-factory* technique.

Models and twins can have rudimentary tests for scopes and their decorating logic. They are read-only tests and similar to what you already do in your Rails apps (hopefully).

Cells and integration tests test your UI. Since business is covered in the operation tests you can stay focused on visual testing here.

```
1 let(:comment) { Comment::Create.(comment: {body: "Awesome!"}).model }
```

```
2
```

```
3 it do
```

```
4   cell("comment/row", comment).must_have_css("li.comment")
```

```
5 end
```

I was playing a lot with different approaches and found the results very satisfying. A good test suite is the guarantee for a good sleep.

A Note On Complexity

Trailblazer introduces a new level of complexity into your application. Where teams previously had to deal with only models and controllers, there's now operations, forms, representers and more. Is this good or bad?

Trailblazer brings something that is missing in Rails: Standards. Standards that go further than table names and rake tasks. It brings guidance for architectural questions and standards that have very clear scopes and use-cases. Where Rails has a wishy-washy “put this into the model, because... skinny controller!” convention Trailblazer clearly identifies the different layers of web applications and provides abstractions.

Trailblazer is no “complex web of indirections” but a layered system architecture that handles many problems of Rails with mature gems and very loose coupling.

It brings more layers to learn but at the same time relieves the conventional “MVC” components and encourages maintainable code by splitting up concerns into different components.

It's a matter of communication to train developers to think in a high-level domain, endpoints, forms, representers, and Cells instead of confusing them by pushing every possible line of code into the model and helpers.

Summary

Trailblazer is very explicit. Instead of letting the framework guess what you want you have to define it - using a simple and consistent declarative language.

High-level domain functions are implemented in operations that can be used as controller endpoints, as test factories, in the console or scripts. HTTP-related code is handled in controllers, operations embrace the business logic.

Every operation has a form object. This is used to deserialize and validate the incoming data. All further logic and persistence happens in the operation, which can use the form object to push data to the model and persist data. Also, callbacks as known from ActiveRecord are moved to the specific operation.

Presentation happens in Rails controller views or Cells. It's your choice which degree of encapsulation you want: Rails views may render models, operations or form objects. Cells decorate models and implement parts of the page.

A third way of presenting is via HTTP APIs. Trailblazer offers you representers to render and parse JSON, XML and YAML including hypermedia and all the cool stuff. Again, representers need to be explicitly defined. Luckily, contracts use representers internally and can be re-used for serialization and deserialization of models.

Twins provide a simple way for reusable decorators and data structuring like compositions. They can be used in any layer of Trailblazer.

And now, let's go and do some actual coding. I'm tired of this lecturing.

Operations And Forms

The Example app

Throughout the course of this book we're gonna implement a real running Rails application called *Gemgem*¹⁵. The concept of Gemgem is simple: Comment on things.

You literally present *things* on their own shiny page. A thing can be just anything, like a book, a band, or a Ruby gem. Besides information about the presented object, the thing page also allows commenting. *Comments* will have a weight which can be either positive or negative. And that's basically it.

Additional features like moderated comments, notifications for comments, a slim in-page admin mode and user authentication will make this book a hopefully interesting read for you, dear reader, as I try to cover everything realistic found in a typical Rails application.

While this might sound pretty tedious and boring let me explain why Gemgem is important to me. Gemgem is actually planned to become a feedback aggregator that matters.

One thing I've been missing in the Ruby world is a place where I can easily collect user voices about gems. As a gem author it is very important to have a constant feedback flow from your users. This is not only beneficial for productiveness, but also stability. For an author, proper feedback can be extremely encouraging and continually motivating to keep working on something. Likewise, reviews and ratings help users to find gems suitable for solving their problems.

Feedback drives a community and its products.

I am aware of awesome pages out there like the [Ruby Toolbox](#)¹⁶. There's numerous other projects that allow commenting and the like. Did anyone say Reddit?

However, I always wanted a web app to keep all feedback about a certain thing in one persistent place and streamline the way feedback is collected and presented. And this is Gemgem.

Going further, I am planning to let Gemgem grab and reference comments about a thing from other pages, making it really simple for authors and users to track what's word on the street.

Preparations

Gemgem can be found online on the incredible Github, and so a working version of this chapter's Rails application [can be found on Github](#)¹⁷!

¹⁵This nifty name comes from my ex-workmate and friend [Garrett Heinlen](#). Our love for Ruby, beers, dance, and maroon-colored shirts has banded us together for life.

¹⁶<https://www.ruby-toolbox.com/>

¹⁷<https://github.com/apotonick/gemgem-trbrb/tree/chapter-03>

I've prepared branches that correspond to concise stages of every chapter. These branches will help you to see the entire application at this stage.

As a trade-off for this I won't walk you through every little code change in this book. We're going to discuss all the relevant steps and conceptual necessities in detail but I'm not permanently gonna explain why I added this `div` tag to that view or how a particular collection of records were found. This saves both of us from wading through a million code lines per page.

Trust me, you're gonna understand how Trailblazer works with Rails. It's gonna be fun. I actually can't wait to jump into the code. Let's do it!

Gemfile

As with every framework, Trailblazer needs to be loaded, and as always, this happens via the `Gemfile`. Here are the relevant lines.

```
1 gem "trailblazer-rails"
2 gem "trailblazer-loader"
3 gem "reform", "~> 2.1.0"
4
5 group :test do
6   gem "minitest-rails-capybara"
7 end
```

The `trailblazer-rails` gem will load the core gem and run some more initialization code for you convenience (line 1). For example, ActiveModel support for Reform is automatically included, and so on.

Since Trailblazer introduces a different file structure we decided a `trailblazer-loader` gem will nicely bridge Rails autoloading and our super primitive, explicit class loading (line 2).

Trailblazer runs with all kinds of Reform versions, and since we're gonna make extensive use of the form gem, I lock it to 2.1.0 which is the newest release by the time of writing this (line 3).

We will do integration tests using Capybara and MiniTest, so I include the `minitest-rails-capybara`, too (line 6). Note that you are free to use RSpec in your own apps, if you prefer that.

File layout

In many tutorials I've seen so far, writing an application starts with implementing a `User` class or table or whatever. The first thing to create then is the sign up page. I hate that.

Even though I know that my application will have users logging in at some point, at this very moment I feel like cranking out the application's actual *domain*: allowing to comment on things. Why would I waste time with a stubborn user registration form now?

This workflow becomes even more significant in a money-driven environment: imagine you were to present your business idea to some lame business dude. This person doesn't care about whether users can sign up using a captcha-backed form or via Facebook login. They want to see what makes your app awesome, they want to see your *domain* in action.

And this is why we will implement how to create things now. Hey, don't get me wrong, I'm not a lame business dude at all. Money for me is printed paper. Nothing more.

In Trailblazer, this development approach is built-in. Trailblazer makes you think about your domain, your business and not about what table is to be migrated and which association needs to be pointing to what join table. That is all stuff to be refined when it becomes a problem.

The first thing we do is we create a new *concept*. The *thing* concept. While we have generators for that in Trailblazer we're gonna do that manually in this chapter. Yepp, some extra workout for you.

I start by setting up the following file and directory layout.

```
1 app
2   └── models
3     └── thing.rb
4   ├── controllers
5     └── things_controller.rb
6   └── views
7     └── things
8       └── index.html.haml
9   └── concepts
10    └── thing
11      └── operation.rb
```

Code for the *thing* concepts goes into a separate directory in `concepts/thing`. We will group all CRUD operations in the `operation.rb` file. I'll come back to that in a second.

Trailblazer uses models, views and controllers the way Rails established them. There is no replacement for controllers, yet. Rails' model layer is just fine as it allows you to use ActiveRecord, or any other ORM. Views can be modularized or replaced using Cells.

This is why the original three "MVC" directories are still there. If this annoys you because you were expecting Trailblazer to make everything different, wait until you see how slim the original layers become and how code gets pushed into new, more appropriate tiers.

The Create Operation

In order to start using our software, let's start with an operation to create a thing. As already discussed in chapter 2, operations are an application's endpoint for the high-level domain. They implement all the public operations a user can perform.

In a Trailblazer project, your high-level domain will automatically evolve. It's a side-effect of moving business code into operations that embrace or define your domain. No one requires you to formulate your complete high-level API right now! You're gonna be surprised how thinking about building applications shifts from "*Where do I add this in which model?*" to writing a new operation whenever you expose a new function to the public.

Persistence

You might have noted that there's an `app/models/thing.rb` file. This is a standard `ActiveRecord::Base` model class.

```
1 class Thing < ActiveRecord::Base
2   has_many :comments
3 end
```

Following the Trailblazer style, it only contains persistence configuration. Here, a `has_many` directive for comments which we're going to use soon. There's no callbacks, no validations and no additional business code in this class - and that is a good thing.

I chose to use ActiveRecord in this book to make it easier for developers to relate to their own project, but you can literally use any ORM framework you want.

Note that there's also migrations that create an initial database. What matters for us is the following snippet, only.

```
1 create_table "things" do |t|
2   t.text    "name"
3   t.text    "description"
```

A thing always has a name and a description. This is pretty amazing.

The Operation Class

Ok. We wanted to allow the user to create a *thing*, right? This is our first high-level domain action that will reside in a fresh `Operation` class. After playing with several file layouts I ended up putting CRUD operations into the `operation.rb` file in the concept's directory. I am talking about `concepts/thing/operation.rb`.

This file will contain all CRUD code for things. Let's have a look at this file.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     def process(params)
4       @model = Thing.create(params[:thing])
5     end
6   end
7 end

```

I usually put operations for a model into the model's namespace. This is why I create a class inside the `Thing` class which results in the global name `Thing::Create`. Reading this you instantly know what is the responsibility of this operation.

Ruby's namespaces

Please note that this absolutely does not tie the operation to the `ActiveRecord` model! All we do here is reusing the model's namespace.

Namespacing helps grouping operations that belong to one *concept*. Putting classes into other classes does not extend, derive, bundle or tie anything to anything. The inner class doesn't know about the outer class, and vice-versa. This is a pure structural Ruby technique that has - admittedly - caused some confusion for new Trailblazers.

Another benefit of namespacing is that you don't pollute the global namespace. Instead of nesting classes as we do it, the operation could also be called `CreateThing`, `ThingCreate` or whatever on the global namespace. In Rails, this would probably end up as `ThingCreateOperation`, which, frankly, looks horrible. Trust me, the namespacing is really helpful for structuring.

Later in the book, we will introduce concepts that don't have a direct 1-to-1 mapping to a model. The "feed" and "follow thing" concepts are a good example for that. Instead of using the model class we will use a concept namespace. Also, we will learn that operations are not limited to CRUD semantics but can implement just anything.

The process Method

Coming back to the operation implementation you will notice that there's only one method present, called `#process`. This is the only requirement that needs to be implemented.

```

1 def process(params)
2   @model = Thing.create(params[:thing])
3 end

```

You are going to write a lot of process methods in this book. They always receive the parameters hash that was passed to the operation from the caller, which could have been a controller action, a test or a console call.

Presently, I simply by-pass any validations and call `Thing.create`. Note that operations require the `params` hash to contain the model attributes under a separate hash key (here `params[:thing]`). This makes it work seamlessly with Rails' well-established `params` concept and allows passing additional environment data into the operation.

Passing the entire `params` hash into the operation might appear awkward. It actually took me weeks and months to figure out what's the best way to provide data to the operation. As usual, the simplest approach wins.

Testing Create

The above code is enough to hook our operation into a controller.

No. No no no! We're not gonna do that until we've written a test for that operation. I know how tempting it is to simply plug it onto a route and see what's happening. However, this book's hidden agenda is all about testing. So let's write a test.

In Trailblazer, business logic is tested by testing your operations. In Rails, this usually is split into controller tests and model tests, making it hard to figure out behavior from tests.

```

1 class ThingCrudTest < MiniTest::Spec
2   describe "Create" do
3     it "persists valid" do
4       thing = Thing::Create.(thing:
5         {name: "Rails", description: "Kickass web dev"})
6       ).model
7
8       thing.persisted?.must_equal true
9       thing.name.must_equal "Rails"
10      thing.description.must_equal "Kickass web dev"
11    end
12  end
13 end

```

Usually, all tests for a particular concept sit in the same directory. This groups tests for cell, operation, representer and twin and again embraces the component structuring. Tests for our operations go into `test/concepts/thing/crud_test.rb`.

This test first calls the `Create` operation and passes in the parameters that define this test case. Since the operation *call style* will return the operation itself I invoke `#model` on it to retrieve the created model (line 4-6).

The following three lines assert that the created model was populated and saved correctly. This is nothing too fancy (line 8-10).

You might find my test style funny because I barely use the test framework's sugar. All I do is grouping test cases using `describe` containing one or more `it` blocks. The `it` block embraces multiple assertions.

While you're free to use whatever test framework and style you want there's several reasons I do write my tests like this.

Firstly, I hate indentations. Using `describe` was originally designed to create reusable environments, and that is great. However, the more levels of `describe` you nest to achieve a reusable setup the more you lose track of why you actually nest blocks and what is being tested. Or, in other words: The less I nest the better readable my test code gets and the easier I find it to restructure blocks.

The other thing I've seen lots of tests incredibly overloaded with `its`. In my above example, I could easily split the one block into three separate `its`. This not only creates unnecessary noise, this is just wrong. My particular test is about asserting the result of one single `create` operation - and that means I explicitly do *not* want to run this operation for every assertion.

Regardless of the outcome of our test style battle: the operation is now good to go. Let's hook it into a controller.

Controllers

To use an operation the simplest way is to use Trailblazer's `run` method in controllers. `run` is imported automatically by the `trailblazer-rails` gem.

```
1 class ThingsController < ApplicationController
2   def create
3     run Thing::Create
4   end
5 end
```

This runs the operation and then returns flow control back to the controller (line 3). It's probably easier to understand when I show you what happens inside `#run`. Here's what basically happens when you use `run` in a controller.

```
1 Thing::Create.(params)
```

As you can see, `run` automatically passes the `params` hash into the operation invocation. I know, this looks as if you could easily do it yourself. In later chapters we'll see how helpful that is when composing parameters before passing them to operations. For instance, `run` can merge the current user into `params` to allow an operation to handle authorization, too.

It is important to mention here that you can still use the existing controller rendering. Trailblazer comes with Cells to encapsulate parts of the page, but we still use global views for controllers. Moving code from the controller into operations doesn't limit you - it is a structural improvement.

```
1 def create
2   run Thing::Create
3   render action: :show
4 end
```

See, you can invoke an operation and still do conventional Rails view rendering.

Running an operation with `run` comes with another benefit. Every operation has exactly two states: success or invalid. The block style for `run` integrates these two different scenarios with your controller.

```
1 def create
2   run Thing::Create do |op|
3     return redirect_to op.model
4   end
5
6   render action: :new
7 end
```

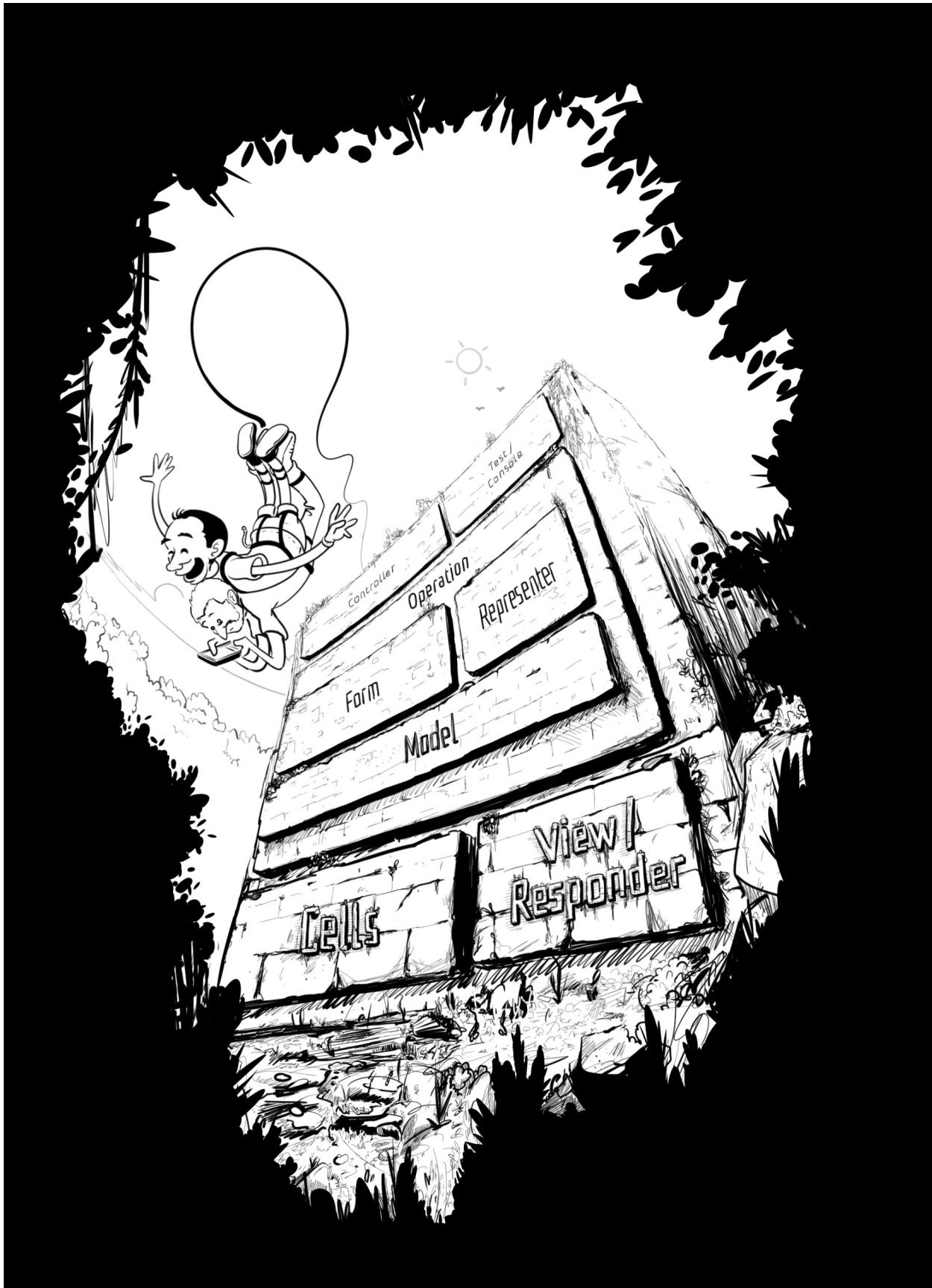
The block is only run when the operation was invoked successfully, resulting in a redirect to the freshly created thing (line 3). Since that block returns from the `create` method, the remaining code is not run. In turn, given that the operation is invalid, the code below the block is executed. That will re-render the form and show possible validation errors (line 6).

Beside the fact that this is an extremely handsome controller action, two things are a bit awkward at this current moment: We don't have a new action and view, yet, and even if we had one, how would that look like, and how would the form to create a new thing be rendered?

The second point is, and I don't know if you remember it, but our operation doesn't do any validation checks and the like, so how would it know whether or not it was run successfully?

The answer is: it doesn't. Without validations, the operation always "thinks" it was run successful. While this reflects an extremely positive attitude towards life this is not quite what we want for our business rules.

Our next step is to add validations to the create operation. In Trailblazer, this happens by adding a form object - exciting times!



Reform: Form Objects

In order to add validations to the create operation, I would like to introduce you to Reform's form object, first. It won't take long and when we see how operations and forms integrate you will understand why I am insisting on talking about Reform now.

Please, for two minutes, forget about the operation we just built and imagine we were to implement a form for creating things and its processing code directly in a controller.

Reform allows you to define fields and validations in a separate class - the *form* class. In Vanilla Rails, validations go directly into ActiveRecord classes and make it extremely hard to reuse validations in different contexts.

Here's the Reform class I'd write for the Thing::Create workflow.

```

1 class ThingForm < Reform::Form
2   property :name
3   property :description
4
5   validates :name, presence: true
6   validates :description, length: {in: 4..160}, allow_blank: true
7 end

```

This class is pretty straight-forward, I believe. Fields are defined using `property`, optional validations can be specified using the well-known `validates` method with all the goodies known from Rails.

Form objects can validate input and mediate the sanitized data to models.

Furthermore, a controller action could also use this class for rendering a form. Presenting the form could happen using `simple_form` or any other form builder gem. In order to take advantage of the form builder we need to instantiate the form first.

```

1 class ThingsController < ApplicationController
2   def new
3     @form = ThingForm.new(Thing.new)
4   end
5 end

```

When creating a form object, you always need to provide a model to the form. Here, this is a new Thing instance. I assign the form object to the `@form` instance variable to use it in the view (line 3).

All additional abstraction layers in Trailblazer usually decorate - or wrap - a model. This is a pattern found in forms, cells, representers and twins.

Check out the corresponding `app/views/things/new.html.haml` view and how I use the form object with `simple_form` as an intermediate object between presentation and persistence.

```

1 = simple_form_for @form do |f|
2   = f.input :name
3   = f.input :description
4
5   = f.submit

```

See how the form instance can then be used directly with a form builder? The latter doesn't even know it is presenting a Reform object, it still thinks that this is an ActiveRecord model.

This view will render a cute form to create new things for Gemgem. After filling it out, clicking the submit button will hit the controller's `#create` action where the form input is validated and processed.

The following snippet outlines how such a create action could look like.

```

1 def create
2   @form = ThingForm.new(Thing.new)
3
4   if @form.validate(params[:thing])
5     @form.save
6     return redirect_to @form.model
7   end
8
9   render action: :new
10 end

```

While this looks pretty familiar please note how we make heavily use of the form object.

To validate the submitted input we use the form's `validate` method that accepts a hash of parameters and then in turn uses the validators we defined earlier (line 4). When validating the input, Reform does *not* write anything to the model, yet, cleanly separating the persistence layer from the business.

After a successful validation the form can update attributes and save the wrapped `Thing` model using `save` (line 5). What an incredible API - both methods do exactly what their name says!

In case the validation failed the form provides a Rails-compatible list of errors. This can be retrieved using `@form.errors`.

Check out the controller code surrounding the form invocations: this is nothing more than orchestrating instructions that delegate view rendering and redirection according to the form's state.

And that's Reform. Let me summarize what we just did.

1. We defined a form with properties and validations.
2. When instantiating forms you need to pass an existing or a new model into the constructor.
3. The form instance can then be used in the view to render a HTML form, e.g. in the `new` action.

4. Likewise in `create` or `update`, processing the submitted data works with the form's `#validate` method.
5. Dependent on the validation result, the form can also push validated data back to the model and save it.

Ok, we got an operation to embrace the entire process of creating a thing. We got a form object as *a part* of that to validate and process the incoming data. Now, how do these two play together? Do you have to manually create an operation and a form and somehow use the form inside of the operation, or what?

Of course not. You could do that manually, now that you understand the principles of both pattern. However, the form object is integrated into the operation.

Operation's Form Integration

Instead of having to define the form separately you can do that in the operation. In Trailblazer, you usually have a form object and an operation combined, making it surprisingly simple to use the form's deserialization and validation mechanics in the operation.

Let's check out how an operation with a form object looks like.

```

1  class Thing < ActiveRecord::Base
2    class Create < Trailblazer::Operation
3      contract do
4        property :name
5        property :description
6
7        validates :name, presence: true
8        validates :description, length: {in: 4..160}, allow_blank: true
9      end
10
11     def process(params)
12       thing = Thing.new
13
14       validate(params[:thing], thing) do |f|
15         f.save
16       end
17     end
18   end
19 end

```

The familiar `Thing::Create` operation gets extended with a form. Additionally, I changed the processing code to consider validations.

In operations, a form is called *contract*. Basically, I copied the form's content into the `::contract` block (line 3-9). This DSL method does nothing more but creating a Reform class behind the curtain for you. It also helps dealing with inheriting contracts to other operations, this is why I originally introduced the `::contract` method.

That said, it becomes obvious that you may use any of Reform's goodies in the contract block. Again, this block is simply a Reform class, you're free to use advanced validations or nested forms, which we're gonna learn about later.

After adding the form (or contract) I also modified the `#process` method in the operation. The original call to `Thing.create` got replaced and I now use `Thing.new` to instantiate a fresh model for the create (line 12).

Creating the form object, populating it with the incoming data and validating the input all happens in the `validate` invocation (line 14). The required arguments here are the actual parameters from the form submission and the model the form will wrap.

After verifying the input, the `validate` method invokes the passed block *only* if the validation was successful (line 15). The form that gets yielded into the block offers me a convenient way to update attributes and save the model - by calling the form's `#save` method I avoid doing just that manually.

Model Semantics

Operations are designed to cover an entire application's high-level domain. Looking at conventional Rails projects (or all kinds of applications) you will see that more than 85% of your domain is CRUD logic: creating and updating objects, whether they are persistent or not, is the main task of every project.

Trailblazer comes with CRUD, or model semantics for operations that standardize a few steps so you don't have to think about them anymore.

Please, have a look at our operation after using the `Model` module.

```
1 class Create < Trailblazer::Operation
2   include Model
3   model Thing, :create
4
5   contract do
6     property :name
7     property :description
8
9     validates :name, presence: true
10    validates :description, length: {in: 4..160}, allow_blank: true
11  end
12
```

```

13  def process(params)
14    validate(params[:thing]) do |f|
15      f.save
16    end
17  end
18 end

```

This is getting really simple.

I first include `Model` into the operation, this will resolve to and load the `Trailblazer::Operation::Model` module (line 2).

Instead of creating the model manually I tell the operation what to do (line 3). Using `::model` will instruct the operation what kind of model to instantiate for processing.

This greatly simplifies the `#process` method (line 13). The model is now created automatically in the invisible `#model!` method that we're gonna discuss in a few chapters. Also note that I don't need to pass the model into `validate` anymore (line 14). Only the parameters go in, the rest is handled by the `Model` module which I find extremely convenient.

Fantastic. Even if I didn't manage to make you entirely understand how things work together, you get a pretty good idea about what the operation does and is supposed to do. The code we just wrote can be used for three different use cases.

1. Our `Create` class validates and processes input and persists a new `Thing` model populated with sanitized input, given the validation was successful.
2. It can also be reused to render a form, or more precisely, to help the form builder to render a form. This is helpful for the `new` action where you allow users to create a new thing by filling out a form.
3. As if this wasn't enough, the operation can also be used to render an invalid form. This works like 2. only that the form builder (or HTTP API code) can access validation errors from the contract, too.

"Wait - this is not SRP, this is too many responsibilities in one class!" you will think now. Your point about the operation's inflated responsibility scope is valid. But also wrong.

Keep in mind that an operation is composed of more than one object. An operation maintains a form object which is completely decoupled from the operation itself - the form doesn't even know it's being used in a Trailblazer environment. A Reform object comes with the ability to validate input, save models and present errors, and the operation acts as an orchestrating dispatcher, only.

Don't confuse reusability with responsibility. Where Rails usually exposes one physical object to handle everything (also known as the *model*), Trailblazer has a fine-grained object design. This

allows cleanly handling several steps of a typical workflow with “one” object: The API simply dispatches to internal objects which cover one responsibility, only. I am going to talk about SRP and the misunderstandings with this concept in a later chapter.

If you’re still sceptical, I invite you to have a look at [Operation’s code¹⁸](#) - it is surprisingly simple and delegates more than it actually implements.

Next, I want to discuss how we can reuse the operation for rendering forms to allow users working with our upcoming website.

Rendering Forms

You will have noticed that I started this chapter with a workflow that might seem counter-intuitive. Instead of programming the form to create a thing I actually gave precedence to the processing logic, first.

This is good in two ways: Most of the hard work is already done, and we need to write very little code to implement the form rendering now.

The controller action to display the UI for this will go into the `#new` action of `ThingsController`, following a Rails conventions that actually has made my life easier.

```
1 class ThingsController < ApplicationController
2   def new
3     form Thing::Create
4   end
5
6   def create
7     # ..
```

This is not a lot of code. I told you, the hard work has been done already. Now, what happens here?

Calling `#form` in the controller will instruct the operation to only instantiate its model without running any processing code. The operation will create or find the respective model for you as this reuses the same mechanics from the creating process we discussed earlier.

After the model is created or retrieved, the operation instantiates its contract, using the model as the *form model*. This allows to use the Reform object outside of the operation, e.g. in combination with a form builder.

The missing piece now is how the form is made available to the controller. And you guessed right, the `#form` helper assigns the controller instance variable `@form` which can then be used in the view.

Why don’t we check out the controller’s view in `app/views/things/new.html.haml` which is rendered automatically in the aforementioned action.

¹⁸<https://github.com/apotonick/trailblazer/blob/master/lib/trailblazer/operation.rb>

```
1 = simple_form_for @form do |f|
2   = f.input :name
3   = f.input :description
4
5   = f.submit
```

That does not only look exactly identical to the code we had a few pages ago, that is the exact same snippet. And I am absolutely not trying to fill pages with fluff. The fact that an operation's contract is a Reform object makes this part very straight-forward. You can use the operation's form just like you did it directly with Reform. Form builders love Reform.

Here's what we have so far.

1. In `ThingsController#new`, the operation finds or creates the model. After creating the form object we can use the latter to render a form in a controller view.
2. The user fills out the form and clicks submit.
3. Submission of the form will hit `ThingsController#create` which simply delegates work to the `Create` operation.
4. Again, a model is created. This time, the operation runs `#process`, validates the input and updates and saves the model in case of valid input.

The controller then redirects to the new thing's URL.

5. Given the input was not adequate the operation will mark itself as invalid. The controller won't redirect but re-render the new view. Since not only the `form` controller helper but also `run` assigns the `@form` instance variable this will render the operation's form - this time, it will also display validation errors!

How is the form builder in the `new` view able to render erroneous fields? How does it know about validation errors that happened somewhere in the operation, in its form?

The answer is very simple. The form exposes a form builder compatible `#errors` method. When rendering, the builder asks the form about errors, which happens via this method.

Awesome! We implemented an entire workflow of displaying a form, processing, and handling errors. I am a bit proud of you. But I'm also concerned since we barely test any of this.

Let's spend the next two pages on how to test our controller actions and the operation with its form.

Testing Validations

We extended the `create` operation to use a form, allowing to validate input instead of blindly persisting it. Looking into `thing/crud_test.rb` we already assert the happy path. We now need to test an invalid scenario.

```

1 it "invalid" do
2   res, op = Thing::Create.run(thing: {name: ""})
3
4   res.must_equal false
5   op.model.persisted?.must_equal false
6   op.contract.errors.to_s.must_equal "{:name=>[\"can't be blank\"]}"
7 end

```

To test invalid input I use the operation's `#run` method as it returns the validation result along with the operation instance without raising an exception in case of an error (line 2).

In the following line I assert the result is false (line 4) and the model wasn't persisted (line 5). I wouldn't test the latter normally, but since this is a book I want to look like a super-correct developer.

What is important is the last line of the test. Accessing the contract's `errors` object, converting it to a string and asserting the error messages is a fundamental test that assures my validations actually work (line 6).

As we have complex validations in this operation, we also must test the `length` validator from our `description` property.

All kind of validation tests are tested via the operation that contains the respective form object. Neither does it make sense to test form objects directly (you could do that, though) nor are we keen to write slow, opinionated controller tests for all edge cases. Since the operation keeps our business logic this is the perfect place to assert validity.

```

1 it "invalid description" do
2   res, op = Thing::Create.run(thing: {name: "Rails", description: "hi"})
3
4   res.must_equal false
5   op.contract.errors.to_s.must_equal \
6     "{:description=>[\"is too short (minimum is 4 characters\")]}"
7 end

```

Again, all I do is provoking a validation error, this time with a too short description. This test is very verbose and later we will learn how to use matchers for all kinds of generic validation tests. They greatly improve readability and save brain-power invested into writing those validations tests. For now, however, I decided it's good to show you the mechanics on a lower level.

Tests like that make me sleep at night. What I love about operation tests is that they are incredibly simple and fast, both to write and execute. The style is always very simple because of the unified

Operation API. Testing various invalid scenarios is a walk in the park. This is different to Rails' model tests which only test parts of your business code.

Operation tests will automatically test all your domain logic - an operation *is* your domain layer.

Also, I always hated writing controller tests. They are slow and clumsy. And when you look at the implementation of `ActionController::TestCase` you will find out why. An insane amount of code is used to setup a leaky test environment that is the opposite scenario of production code.

The result will be tests that might pass but the very same code still fails in production.

Always remember: the more code you need to setup your test environment the more likely it is to break on real servers with real users and real, merciless requests.

Testing Controllers: Integration Tests

The controller is incredibly slim in a Trailblazer setup, look at the actions in `ThingsController` implementing new and create. This is not because we code hide the way `inherited_resources` does it. This is because the code now sits in an operation which is completely decoupled from the controller - and thoroughly tested.

When it comes to controllers, I always simply test the wiring. I call this a *smoke test* as it doesn't assert any edge cases and only runs on a very superficial but yet efficient level.

I would like to quickly run through `test/integration/thing_test.rb` now. As you can see, I structure integration test by concept and semantic, since some of the tests will grow during this book, just as some stomachs will.

```
1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     visit "/things/new"
4
5     # invalid.
6     click_button "Create Thing"
7     page.must_have_css ".error"
```

The new test class I derive from `Trailblazer::Test::Integration`, which is provided by the `trailblazer-rails` gem (line 1). It automatically allows us to use the Capybara gem for realistic integration tests with selector matchers like `must_have_css`.

I first let render the form by browsing to the *new* page using the `visit` helper that accepts real paths (line 3). This implicitly tests rendering of the page and will fail if we did something wrong here.

By submitting an empty form and checking for CSS classes in the resulting page, I make sure both routing to the update action, processing thereof, and rendering an erroring form works (line 6-7). The default Simple-form behavior is to render at least one `div` class named `.error` and this is all I test.

```
1 it "allows anonymous" do
2   # ...
3   # correct submit.
4   fill_in 'Name', with: "Bad Religion"
5   click_button "Create Thing"
6   page.current_path.must_equal thing_path(Thing.last)
7 end
```

As a follow up test, I submit the form again, this time, by filling in correct values (line 4-5). Capybara gives me a really nice DSL to achieve just that using `fill_in` and `click_button`. Again, this is also an implicit render test as those calls would fail if form elements can't be found.

Instead of making wild assumptions about what the business logic might have done I simply test whether the endpoint redirects me to the newly created object (line 6). I find the latest model using `Thing.last`. Note that there's better ways to retrieve the operation's result, we're gonna learn that later.

The focus of *smoke tests* is to make sure a minimal set of constraints is met. Smoke tests assert that the controller runs for the happy path and for invalid data - nothing more. Since the business logic is already tested in operation tests there's no need to repeat it in an integration test.

A few days ago I explained integration tests to my friend Jonny as a click test. Before programmers used automatic testing, they would click through common paths of their app after having changed code, without focusing on details, just to make sure the flow isn't broken.

Of course, this is unacceptable for high-quality software, but a controller in Trailblazer is nothing more than a stupid HTTP endpoint. The integration smoke test is sufficient to maintain product quality. Our point is: we tested all edge-cases in detail in our operation tests.

Update Operation

In the last three pages of this chapter I'd love to go through the Update operation for editing and updating an existing thing. Several readers requested this operation to be a bit more complex. Just a lil' bit.

Why not make the `Thing`'s name field read-only when editing? Let's assume we're gonna use the thing's name to generate its unique URL in Gemgem - changing the name would also change that URL, which is no good. The "business" decides to make the name static, once it has been chosen when creating the thing.

This is extremely simple in Trailblazer. Thanks to its explicit, declarative style, all we need to do is re-configure the existing Create operation. Of course, this happens in a subclass. This does not only give us the freedom to fine-tune behavior later, it also assures we do not break any create code: a subclass inherits structure and behavior, but the Create operation is completely decoupled from the Update class - it doesn't even know it got derived.

Here's the updating operation in `app/concepts/thing/operation.rb`

```

1 class Update < Create
2   action :update
3
4   contract do
5     property :name, writeable: false
6   end
7 end

```

And that's literally it.

The `Update` class is derived from the creating operation using pure Ruby. This will *copy* all the operation's code and the contract to the new class (line 1).

To give a new semantic to the operation, I override the old action with `:update` (line 2). This will tell the operation to find the model, not to create a new one. In order to do so, the operation will look for an `:id` in the incoming parameters.

As the “business” wants us to make the name immutable when editing we need to do so in the contract. Luckily, Reform comes with that out-of-the-box and allows us to override the original `name` property to be not writeable anymore (line 5). This will simply ignore the name field when processing the form.

Testing: Update Operation

Maybe looking at the respective test makes it easier to understand what we just did.

```

1 describe "Update" do
2   let (:thing) do
3     Thing::Create.
4       (thing: {name: "Rails", description: "Kickass web dev"}).model
5   end

```

First, note how I reuse the `Create` operation as a test factory (line 3-4). This is really one of my favorite features in the Trailblazer architecture! This will always give you a valid application state in your test cases that is as close to production as possible.

```

1 describe "Update" do
2   # let (:thing) ...
3   it "persists valid, ignores name" do
4     Thing::Update.(  

5       id:      thing.id,  

6       thing: {  

7         name:      "Lotus",  

8         description: "Simply better.."  

9       }  

10     )

```

I then invoke the Update operation with a new name and description (line 4-10). In addition to that, I also have to pass in the id of the object to be updated (line 5). The operation will use that id to find the actual model.

```

1 it "persists valid, ignores name" do
2   # ...
3   thing.name.must_equal "Rails"
4   thing.description.must_equal "Simply better.."
5 end

```

As you can see, even though I sent in malicious data trying to change the name, it remains the original title (line 3). Reform simply ignores it.

Currently, this test is enough to cover all eventualities for the update operation. Since everything else is inherited, we don't need to test it as we didn't change anything else.

Controller

The controller's edit action for rendering the edit form is almost identical to the new action we wrote a few minutes ago.

```

1 class ThingsController < ApplicationController
2   # ...
3
4   def edit
5     form Thing::Update
6
7     render action: :new
8   end

```

Note how I use `Thing::Update` in this action, as we want to display a form for an existing model. What happens here is `form` will automatically pass the entire `params` hash to the operation. This hash contains an `:id` field per Rails convention and allows the update operation to find its model (line 5).

Given you were viewing the URL `http://localhost:3000/things/1/edit` this would translate to the following invocation inside of `form`.

```
1 Thing::Update.present(id: 1)
```

And this is exactly the information the update operation needs to retrieve the edited or updated model.

The corresponding `update` action in the controller called when submitting the edit form works likewise and doesn't need further discussion here.

The Edit Form

We want to reflect the read-only `name` field in the edit form visually. Form fields can be disabled and, even better, `simple_form` comes with the `:readonly` option for just that. To maximize your excitement, Reform knows whether or not a field is writeable. Now, let's put this all together in the `new.html.haml` view that we render in the new and edit action.

```
1 = simple_form_for @form do |f|
2   = f.input :name, readonly: @form.readonly?(:name)
3   = f.input :description
```

That's the same form we had earlier, with a minor improvement. I query the Reform object for the read-only state of the `name` field (line 2). This will render a normal input field when used in new context, as it used to be. However, since the update operation sets this field to read-only, this will display a disabled input field in edit mode - just as the "business" requested it!

Testing Form Rendering

We did some small changes to the form. In edit context, the `name` field is read-only. When creating a new thing, however, this field must be editable. I insist on adding that to our controller tests where we test the rendering for now.

I add those tests to `test/integration/thing_test.rb`.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     # ...
4     # edit
5     thing = Thing.last
6     visit "/things/#{thing.id}/edit"
7     page.must_have_css "form #thing_name.readonly[value='Rails']"
8   end
9 end

```

This tests the `edit` action in the `things` controller. I assure that the form field for `name` is really rendered as read-only using `must_have_css` (line 7). Since the `simple_form` gem adds a `readonly` class to the input field, we can easily test that.

Testing the edit form is one thing. However, we also have to assure that we didn't mess up the form we built earlier, that's rendered with the same view. The form for new things definitely needs a writeable `name` field.

I add some quick lines to the very same test case.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     visit "/things/new"
4
5     page.must_have_css "form #thing_name"
6     page.wont_have_css "form #thing_name.readonly"

```

The test makes sure the `name` field is present but not read-only in a create context (line 6).

Reform and strong_parameters

Protecting mass assignments from unwanted input was long handled with hand-made exclusion logic or using `attr_accessible`. This was a source of problems because models were used in different contexts but the attributes could only be defined once, on the class level.

Rails (or, DHH himself) then introduced `strong_parameters` which takes the definition of unsolicited parameters to the controller. While this definitely is an improvement, I still find it unsatisfying. There's no way to add additional semantic and behavior to deserialisation when processing a form. Also, this logic is now coupled to the controller *and* to the model, as there's usually extra code in the model to rename or move parameters coming from the controller.

To me, this looks like a half-baked fix on top of a leaky implementation for input deserialisation and processing. In fact, there is no abstraction for deserialisation in Rails. Once you pass the `params` hash

into `create` or `update_attributes` it's out of your scope what will happen. Especially in a nested model setup this is extremely frustrating.

`strong_parameters` makes you think you can control the way parameters are deserialized - which turns out as an illusion once things get a bit more complicated.

Again, this a result of Rails' monolithic architecture and its low degree of encapsulation. Instead of simply extracting form logic into a form object, the deserialisation is distributed between controller and model.

Reform doesn't need `strong_parameters` at all. When defining the form class, with all its properties and quirks, the form knows what parameters to process and what to ignore. A form embraces the entire workflow of, well, a form and all the work associated with it.

Remember how we set the `name` field to read-only and that made everything work for updates? That's the benefit of abstraction and explicit code as found in Trailblazer.

Hey, I'm really proud that we made it that far. We implemented the entire Trailblazer workflow for adding and updating things with operations, models, tests, UI and lot of chat. Cool! In the next chapter, I want to spend a bit of time in the view layer. We're gonna learn how to use Cells to encapsulate parts of the views to objects, making them reusable, simpler to test and better to work with.

Cells

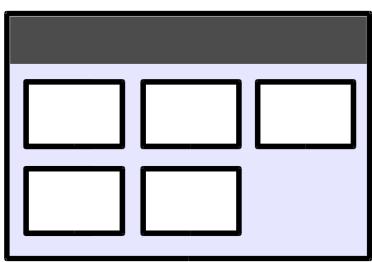
After having implemented an operation with a form to create and update *things* I'd love to talk about views for a brief chapter. I know, we're all keen to learn more about deleting things, adding comments, and all that domain logic, but hang on. The view layer is where you win the business.

Even though my frontend knowledge is close to zero, I love to put a nice UI into place as early as possible. This does not only help discussing and evaluating features and usability, it also makes it look as if you're actually working really hard on stuff.

A working version of this chapter is - as always - available [in the repository¹⁹](#).

Rails Views and Encapsulation

Do you remember what I said in the introduction chapter a while ago? Basically, I am writing this book and all that because I wasn't satisfied with how Rails makes you do view components.



The homepage displaying five recently added things.

Suppose we wanted to display the last nine recently created things on our homepage. And, assuming we had less than nine things to show, they would still wrap up properly in our grid.

Given we're using the Foundation grid and Rails this is pretty simple to accomplish. You grab the last nine things, write a quick partial for one box displaying one thing. And then you do something as follows.

```
1 = render Thing.recent
```

Of course, this looks great. Rails has a tendency to beautify complex things and make them look less complicated.

However, this only works if the global partial's name is `things/_-thing.html.haml`. And, going further, this doesn't handle the case that the last thing box needs a CSS class `.end` to make Foundation render the grid properly even when there's less than nine boxes. But, hey, this is really simple to solve since you can write a helper method to "encapsulate" the rendering of nine or less boxes on our homepage, and the helper will also handle the `.end` class properly... somehow.

The view code probably ends up in one helper "method" `render_boxes_on_homepage` - which is actually a global function - to represent a certain part of the UI. In our case, this is the entire *things* grid.

¹⁹<https://github.com/apotonick/gemgem-trrb/tree/chapter-04>

Each grid item is then modelled with a partial. It needs a bit more than just markup, so the code to detect the last grid item goes directly into the partial. Don't tell me you'd write another helper method - it is simply too awkward to push a "private" code concept for our things grid onto the global helper stack.

I am not saying that a bit of logic in views is wrong. What I am saying is that every project I've seen had a terrible view layer, where the data modelling and the implementation just didn't feel right - given that Rails claims itself an "object-oriented framework".

And I get the same feeling now, assuming I had to write partials and helpers to implement that grid. The whole global helper/partial approach never really made sense to me.

Why are we forced to think about views as functions rendering arbitrary templates, templates without any interfaces and access to global state? Didn't we stop doing PHP because of that? Why can't we think about views as *widgets* where a widget, whatever that is, represents a certain part of the page? I'll tell you why. The Rails view layer hasn't been touched in almost ten years. No one saw the need to introduce change into this PHP-inspired stack layer.

View Models

The Cells gem tackles views from the opposite direction. A cell completely embraces a fragment of the UI. That fragment does explicitly *not* have access to global state - it is implemented as a separate object.

A cell is like a *widget*. Logic and templates needed to present that part are encapsulated in this widget. Any dependency required from the environment, say, the current page in a paginated list, has to be passed in from the caller. This creates an interface for a view.

Which gives us a few advantages over Rails.

- The controller or the view rendering a cell doesn't know anything about the cell, and the cell doesn't know who's rendering it. The effect is called *reusability* as we could now render this cell just anywhere in our app - as long as the input is the same, the output will be identical. Don't even think about that with global helpers and partials.
- The interface drastically improves testability. Again, since we can render this cell isolated in a test and assert the rendered markup, the cell will work in any environment that provides proper input.
- Interfaces make us think. The cell is expecting input and you have to provide it. This makes you question how and especially *where* data is collected. Suddenly, there are more places than a clumsy controller and confusing helper functions. It feels more natural, as a cell can be a mini-controller in the real sense of *MVC*.
- Interfaces also help you understand. Even though I've seen "clean" nested helper/partial constructions that pass locals, they still access global instance variables somewhere, making it incredibly hard to understand dependencies. Cells require explicit arguments, there simply is no global state.

- The view architecture Cells encourages also makes it easier to spot bottlenecks in rendering. Now that you model your UI in nested widgets instead of nested helper calls and partials, you can easily turn caching on and off or disable certain parts of the UI.

Modelling the UI with Cells

When I begin encapsulating parts of my UI into cells, I always do the bottom-top approach: I start with the smallest component possible and see how far I get. Then, I might add a container cell to wrap the smaller components. Here, we'll start writing a simple cell that represents one of those grid items displaying one particular thing.



Currently, one box shows the thing title, links to the actual thing page and displays the date when it was first created.

When adding cells, first thing I do is render the cell in the view. Once it looks good, I write a test. This is more motivating, test-first is good for unit tests, with an UI component it is legit to test it visually first, then add automatic testing step-wise.

The root path of our app currently links to the `HomeController` and in the `index.html.haml` view, I render our first cell. Are you excited?

```

1 %h3
2   Welcome to Gemgem!
3
4 .row
5   = concept("thing/cell", Thing.last)

```

Cells packaged in a Trailblazer concept are rendered using the `concept` helper. I already made fun about the irony of a helper rendering a cell in chapter two. You could invoke the cell manually here but the `concept` helper is handy. All it does is find the cell class by constantizing the first argument. This will result in a class lookup for `Thing::Cell`. It then instantiates this cell and passes in the remaining arguments and invokes the cell's `show` method.

So, the `concept` call basically gets translated to something along this.

```
1 Thing::Cell.new(Thing.last).show
```

This is not 100% what happens, but it helps understanding the workflow when rendering a cell. We'll learn more about different call styles in later chapters.

Anatomy of a Cell

Ok, invoking a cell means a class is instantiated and a method on the instance is called. So let's have a look at that cell class which you can find in `app/concepts/thing/cell.rb`. When writing a cell for a concept, I start with a file `cell.rb` in the concept's directory and implement the code there. However, you're free to apply your own naming style or directory structure. You can have as many cells per concept as you feel like and we're going to use a bunch of cells in this book.

```
1 class Thing::Cell < Cell::Concept
2   def show
3     render
4   end
```

Again, I put the cell in the `Thing` namespace, and, again, this does absolutely not bind the cell to ActiveRecord, even though `Thing` is an ActiveRecord subclass. This is purely structuring.

Explicit Rendering

As you can see, the `show` method does nothing else but calling `render` (line 3).

In Cells, rendering is explicit, you *have* to call it, there's no magic rendering as found in Rails controllers. Another difference is that `render` really returns the HTML string. While this is additional code you have to write, this allows you cool stuff like directly returning strings from a cell method, or concatenating views.

```
1 def show
2   render + render(:footer)
3 end
```

In the many years of working with Cells, ten-thousands of users have agreed that the effort of having to add `render`, which is incredible six more characters, does absolutely justify the flexibility you gain. For instance, many users call `render`, `filter` or add wrappings and then return the string.

Anyway, what exactly happens when call `render`? You guessed right, this will look for a view in `app/concepts/thing/views/show.haml`, parse it and return the view string.

Note that we don't have view names like `show.html.haml` anymore. Per design, a cell doesn't know about the request format and whether or not we're supposed to render JavaScript templates or HTML. In Cells, different UI formats are handled with different cells. Again, I want to discuss that in later chapters.

Logicless Views

Cell templates can be any format supported by Rails, or, to be precise, [Tilt²⁰](#). In Cells 4, we replaced ActionView with our own implementation that uses the great Tilt gem to render templates. This not only speeds up rendering but also reduced rendering code to a few lines.²¹

I chose Haml because I like [Hampton²²](#) but you can use ERB or Slim or whatever you feel like.

Now, looking at the view you'll see that there's no big difference to the views known from vanilla Rails.

```

1 .columns.large-3
2   .header
3     = link_to model.name, thing_path(model)
4   .info
5     From #{model.created_at}

```

Ignoring all the markup noise we need to make it look good with Foundation, what matters here is that we can access the Thing instance this cell wraps using the `model` method in the view (line 3 and 5). This is why cells are also called *view models* since they wrap a model (or any kind of object) in order to present it in a user interface.

We also still have access to the wide range of Rails view helpers like `link_to`, as it seems (line 3).

An interesting thing to note here is that neither `model` nor `link_to` are helpers anymore that got copied from the controller into the view. They are both instance methods defined in the cell class itself.

In fact, every method call in the view will be called on the cell *instance*.

This is a fundamental change to the way Rails handles views. In Cells, the concept of “helpers” does not exist anymore. The handy Rails helpers are still available, but these are all instance methods of the cell class.

What does that mean for us now? We could simplify the view to the following beautiful asset.

```

1 .columns.large-3
2   .header
3     = name_link
4   .info
5     From #{created_at}

```

This is the kind of view we're gonna create throughout this book: no logic, no complexity and a well-defined interface. Instead of adding complexity to the view, we call two new methods `name_link` and `created_at` (line 3 and 5).

²⁰<https://github.com/rtomayko/tilt>

²¹Again, I offer to spend a chapter to discuss the Trailblazer gems and their source code, if there's enough interest in the community.

²²<https://twitter.com/hcatlin>

Helpers

We cleaned up the view and move logic back to the cell class. The two newly introduced methods that we call in the view need to be implemented as instance methods.

```

1  class Thing::Cell < Cell::Concept
2    def show
3      render
4    end
5
6    private
7    def name_link
8      link_to model.name, thing_path(model)
9    end
10
11   def created_at
12     model.created_at
13   end

```

As a good object-oriented citizen, I try to expose as few public methods as possible for each class - following my understanding of the *Single Responsibility Principle*. This also applies to cells! And for that reason `show` is the only public method for now.

The methods we invoke in the view don't need to be public, the view is executed in the cell's instance context. Even though I know that they are only "helpers", I make the new methods `name_link` and `created_at` private. This is both me being a stringent interface supporter and to stress the fact that `show` is really the only public method being called by the cell user on the outside.

It indicates a good object design when marking as many methods as possible as private - ideally, you have one public method per class.

In both new methods I simply moved the old invocations from the view into a method body. The cell class itself provides Rails view helpers like `link_to` so we can use them in instance methods. Note that not all helpers are available in cells per default. You need to include the missing modules into the class when you need more helpers.

Exposing helper code as instance method brings another benefit. You don't have to make them private if you want to reuse them. You can have as many public methods in a cell as you want. This gives you an object-oriented helper class that is incredibly simple to use and test.

Properties

Cells, or view models, are designed to wrap arbitrary objects. Exposing properties from the decorated object is a fundamental concept of a view model. In our example, we don't need to write a delegator for `created_at`. A declarative delegation mechanism is built into Cells.

```
1 class Thing::Cell < Cell::Concept
2   property :name
3   property :created_at
```

Using `property` will create the delegation for us and saves typing. Internally, `property` simply defines the method `created_at` in the cell class almost exactly as we did it manually a few minutes ago.

This technique is also helpful for renaming properties and to work with compositions of objects, hash attributes, and more. The `property` method is a concept found in all my gems and we'll learn how they support you with all kinds of data structures in later chapters and when discussing the `Disposable` gem.

Using the Timeago Helper

So far, we display the raw `created_at` timestamp in the view. That doesn't look good. The `Timeago` gem provides a cool helper that makes every timestamp readable for humans. I'm sure every one of you has seen that before, instead of saying "25/12/2014" it will render something like "two days ago". Let's use that in our cell!

You can check this chapter's branch and how the `Timeago` gem is bundled into your application. In order to use it in the cell, the helper module needs to be included. This usually happens automatically in `ActionView` and that is why we have to do it manually in a cell. Remember, a cell explicitly does *not* want to be polluted with all kinds of helpers.

```
1 class Thing::Cell < Cell::Concept
2   include ActionView::Helpers::DateHelper
3   include Rails::Timeago::Helper
```

The `Timeago` helper relies on `Rails`' `DateHelper` so we need to include that in the correct order. This is not Cells being clumsy, this is `Rails`' lack of view architecture and third-party gems assuming that they automatically have access to hundreds of global functions.

Including this helper adds a new method `timeago_tag` to the cell class and we're ready to use it.

```
1 class Thing::Cell < Cell::Concept
2   #
3
4   def created_at
5     timeago_tag(super)
6   end
```

So what am I doing? I override the `created_at` method that has already been defined when calling `property :created_at`. In that new method, I use the `timeago_tag` helper to render the human-readable string (line 5). By invoking `super` I call the “original” method `created_at` that is delegating to `model.created_at` and provides us with the timestamp.

I do this for two reasons. I could have just introduced a new method `created_at_formatted` that internally calls `created_at` and passes it to the `Timeago` helper. However, I didn’t want to change my view and the method name. The second reason is, my plan is to demonstrate Ruby’s OOP features as much as possible in this book. We will meet `super` again very soon.

Again, please note how Cells makes it as simple as possible to move logic out of the view into a real, physical class. Helpers are now instance methods of a scoped object and not a global ActionView object where helper methods need to be copied from the controller into the latter instance.

By changing the view context to the cell itself, no weird copying or including of methods happens in Cells between the “controller” and the view. This is a learning from playing around with ActionView for years and the frustration that came with it.

Let’s quickly sum up what we did so far.

In the main view we invoke our cell. This instantiates a cell object and calls its `show` method which in turn renders a private view. The view grabs data by calling methods back on the cell. Rendering additional markup and calling helpers again happens via the view model. Any logic required in the view lives in the cell.

Rendering Collections

Originally, we wanted to render a maximum of nine things on our home page. We have three ways to do this. The home view could simply iterate the list of things and call `concept` for each item. We could also write another cell that renders all items internally. This is helpful for advanced caching and logic.

The last and simplest is to use Cells’ ability to render collections directly, and worry about everything else later. Let’s do that for now and explore nested cells in a bit.

Rendering a collection of cells works by using the `concept` helper, similar to what you’re used from rendering collections of partials.

The `app/views/home/index.html.haml` view just needs a tiny change to render more than one cell.

```
1 %h3
2   Welcome to Gemgem!
3
4 .row
5   = concept("thing/cell", collection: Thing.latest)
```

That was simple, wasn't it? Calling it with the `:collection` option is enough to make the `concept` helper render a cell for each thing in the array (line 5).

As always, Trailblazer's API is explicit and doesn't guess. If you want to render a collection you need to speak out. This helps preventing "misunderstandings" as we all know them from Rails, where helpers use `is_a?` and other inflection methods to guess what you want to do. Explicit will always win over magic. There's no point in saving a few characters of code while sacrificing our code stability.

Since `Thing.latest` returns the last nine things from the database we're good to go.

There was one more requirement we need to implement, though. If there's less than nine items, the grid needs to know what cell is the last item to make it look neat and properly aligned.

Layout Helpers

What's happening now is that each cell has to check whether or not it's the last one. If yes, the class `.end` has to be added to the cell's `.columns` div. This is a foundation-specific thing I learned by reading the manual!

First, we need to tell the cells what's the last item. This goes into the index view of our `HomeController` for now. Yeah, this is code in the view but we're gonna clean it up later.

```
1 .row
2   - things = Thing.latest
3   = concept("thing/cell", collection: things, last: things.last)
```

As you've already seen, I add another option `:last` to the cell rendering call where I pass in the last item of the `things` collection. Note that `:last` is a generic option and Cells doesn't know about it until we make it aware of this.

That means we have to program this "layout helper" on our own. However, Cells makes that fun! Since we have to set CSS classes in the cell view, I change the view code slightly. This is back in the `Thing::Cell` and its view `app/concept/thing/views/show.haml`.

```

1 %div(class=classes)
2   .header
3     = name_link
4   .info
5     Since #{created_at}

```

What I do now is I no longer pass all the classes statically to the top div but I use a HAML options hash. In that hash, I call the `classes` method that provides the required CSS classes (line 1).

Additional Cell Options

Remember, any method called in the view is delegated back to the cell, so let's implement this.

```

1 class Thing::Cell < Cell::Concept
2   # ...
3   def classes
4     classes = ["large-3", "columns"]
5     classes << "end" if options[:last] == model
6     classes
7   end
8 end

```

While I always add the `large-3` and `columns` class, the `end` class is only pushed onto the class stack when the current cell instance wraps the last model. By comparing the cell's `model` with the `:last` option we pass into the collection rendering I'm able to find out whether or not the cell is the last (line 5).

I should note that additional options passed to the `concept` call are passed to *all* rendered cells and made available via the `options` method (line 5). This is extremely helpful to pipe generic configuration to one or multiple cells as this works with collections and when rendering a single cell.

Assuming we'd invoke a cell as follows.

```
1 concept("thing/cell", thing, border: 1, show_image: true)
```

We can now access the additional options in the cell via the `options` method.

```

1 def show
2   options #=> {border: 1, show_image: true}
3 end

```

Allowing to pass arbitrary configuration into cells, modeling cells around domain and data objects and moving view code into a separate layer, the cell class, introduces a new thinking of how to write views.

When using Cells, developers have continuously reported that view models feel more natural and the encapsulation does absolutely not block them from a rapid development of features. The opposite is the case. It not only gets easier to reuse components in other pages of your application, it also simplifies rock-solid tests for widgets.

Speaking of tests - why not write some assertions to make sure our cell really does what we want?

Integration Tests

In a vanilla Rails setup it is impossible to test view components in isolation. That is because view components do not exist. Writing tests for dashboards or overview pages like the one we just did happens via integration tests that render the entire page.

This is good on one hand. A full-stack test makes sure everything really works and allows simple HTML assertions in order to do so. However, on the other hand, if we decide to move the view component, we need to move the test. Also, integration tests are slow and clumsy as they require a lot of setup that is not really related to our widgets.

I want to show you both ways now. Let's write an integration test and then test the cells in isolation. I usually write full-on tests that cover edge-cases with encapsulated cells tests. Asserting the entire page works happens with a very simple integrational smoke test - similar to how we did it for operations and controllers.

By demonstrating both ways you can decide yourself how to handle this task.

Check out the integration test in `tests/controllers/home_controller_test.rb`.

```
1 class IntegrationTest < ActionDispatch::IntegrationTest
2   it do
3     Thing::Create.(thing: {name: "Trailblazer"})
4     Thing::Create.(thing: {name: "Descendents"})
5
6     visit "/"
7
8     page.must_have_css ".columns .header a", "Descendents"
9     page.must_have_css ".columns.end .header a", "Trailblazer"
10    end
11  end
```

In the first two lines I create our things fixtures. Note how we do that with operations. A major goal of Trailblazer is to not use leaky ActiveRecord-based factories anymore. But I said that twice already.

I then grab the home page (line 6) and assert that both thing cells were rendered properly. I am not good with CSS selectors, so I make sure that there's only one `.end` class, and this has to be on the oldest thing which is called "Trailblazer" (line 10). Remember, the thing cells are rendered in reverse order.

By asserting that there's only one `.end` class we explicitly say that the most recent thing "Rails" doesn't have this class. That is correct since only the last cell in the grid should render this class.

Full-stack tests like this are extremely slow and that doesn't have to be! We can test the cell in isolation and just write a smoke test for the controller instead of a full-blown integration test.

Cell Tests

The cell test goes to `test/concepts/thing/cell_test.rb` as I often group several cell tests into one file. A cell test gives us the same API that we use when rendering a cell.

Another cool feature is that Cells automatically provides Capybara tests if this beautiful gem is installed. Let's have a look.

```

1  class ThingCellTest < Cell::TestCase
2    controller ThingsController
3
4    let(:rails) { Thing::Create.(thing: {name: "Rails"}) }
5    let(:trb)   { Thing::Create.(thing: {name: "Trailblazer"}) }
6
7    subject { concept("thing/cell", collection: [trb, rails], last: rails).model }
8
9    it do
10      subject.must_have_selector ".columns .header a", text: "Rails"
11      subject.must_not_have_selector ".columns.end .header a", text: "Rails"
12      subject.must_have_selector ".columns.end .header a", text: "Trailblazer"
13    end
14  end

```

Markup content in a cell test can be rendered just as we know it from views or controllers, using the `concept` helper (line 7). The returned string is ready to be used with Capybara matchers, which makes it very simple to test selectors and content.

The rest of the test is basically identical to the integration test and makes sure classes are assigned properly (line 10-12).

One thing to mention is that I use controller `ThingsController` to configure a controller that needs to get passed into the cell (line 2). This is unfortunate, and while the cell is completely decoupled from Rails, the URL helpers from Rails need a controller dependency to operate properly. By configuring what is gonna be the “parent” controller, the helpers will work.

Given that we assert rendering details in the cell test now, I’d convert the integration test into a smoke test. I want to talk about that a bit later, though.

One thing I absolutely hate about this test we just wrote is that it replicates a lot of logic from the controller view.

```
1 concept("thing/cell", collection: [trb, rails], last: rails)
```

This is a classic source of bugs: even though it’s “only” a collection we pass into that `concept` call, and we “only” specify the `:last` parameter, this will break at some point. Currently, we’re assuming the caller knows the entire API of our cell which consists of a very special `:collection` and the `:last` option that has to be set properly.

Nesting Cells

There’s two ways to cope with this. We could either wrap the entire `concept` call into a helper, but then we’d have to write an ugly helper test. Or, and that is the way to go, we encapsulate all knowledge about how the things grid works in another cell. Another cell that composes the grid.

This might sound like a bit of overhead now, but as soon as we hit problems like displaying search results, caching and performance techniques in later chapters, we will learn why this was a good choice.

In order to replace the quite complex cell invocation code with something simpler, we need to go to the controller view `app/views/home/index.html.haml`. I change this file as follows.

```
1 .row
2   %h3
3     Welcome to Gemgem!
4
5 .row
6   = concept("thing/cell/grid")
```

Instead of collecting the latest things in the view and invoking the collection rendering, all I do now is calling the new cell `Thing::Cell::Grid`. No data is passed into that cell - and this in turn means the cell itself needs to accumulate necessary objects.

To understand what’s going on here we need to look into `app/concepts/thing/cell.rb` and inspect the new cell class.

```
1 class Thing::Cell < Cell::Concept
2   # original class code...
3
4   class Grid < Cell::Concept
5     def show
6       things = Thing.latest
7       concept("thing/cell", collection: things, last: things.last)
8     end
9   end
10 end
```

The new `Grid` cell is nested into the original cell's namespace, resulting in a fully-qualified class name `Thing::Cell::Grid`. This might seem counter-intuitive as the grid cell renders the container cell, but is physically nested in the inner cell class.

Don't get confused! Similar to the way we nest operations in models, this doesn't create any dependencies between the classes. To me, the grid cell is a feature added on top of the existing code, that's why I stuff it into the existing cell's namespace.

The grid cell is simply a stand-alone class derived from `Cell::Concept` (line 4). All we did was implementing the `show` method and copy the code from the controller view into that very method (line 6-7). Note that you can easily render cells in other cells by using the `concept` helper (line 7).

Rails and MVC

The logic to aggregate the most recently added things is now located in the new grid cell.

And this is another shift in the classy Rails architecture. You might wonder whether or not it's ok to let a view component collect data. Shouldn't that be done in the controller? And you're right! The view shouldn't know about how data is collected.

However, a cell *is* a controller itself. Unlike Rails helpers, a view model is a real MVC component featuring a self-contained controller that renders a view. It is our decision as software designers whether we push data-collecting logic into a cell or let the HTTP controller or an operation take care of this.

In our case, this is a purely view-related issue. The grid is a UI feature that is most probably not included in a document-based HTTP API. For that reason, we can safely move all logic into the cell.

Always ask yourself: is the logic needed in two places, namely the UI and the HTTP API? Then, code needs to be made reusable in either a twin, a PORE or an operation. However, if it's solely used in the web user interface, this code can be placed directly in a cell.

Final Test Setup

Now that we have a container widget in place to hide implementation details from the caller, we can adjust our cell tests. For now, I consider the `Thing::Cell` class a private view model. In our application, the only public entry point for rendering things is `Thing::Cell::Grid`, so let's adjust our cell test.

```

1  class ThingCellTest < Cell::TestCase
2    controller ThingsController
3
4    before do
5      Thing::Create.(thing: {name: "Trailblazer"})
6      Thing::Create.(thing: {name: "Rails"})
7    end
8
9    subject { concept("thing/cell/grid").to_s }
10
11   it do
12     subject.must_have_selector ".columns .header a", text: "Rails"
13     subject.wont_have_selector ".columns.end .header a", text: "Rails"
14     subject.must_have_selector ".columns.end .header a", text: "Trailblazer"
15   end
16 end

```

This is the same test as before, the only thing that changed is that we invoke the `Grid` cell now (line 9).

After asserting all eventualities for the grid cell, we no longer need to do that in the `HomeController` test. As promised earlier, this get degraded to a simple smoke test.

```

1  it do
2    Thing::Create.(thing: {name: "Rails"})
3
4    visit "/"
5    page.must_have_css ".header a", "Rails"
6  end

```

The smoke test makes sure the cell is included and rendered, nothing else matters.

The cell Helper

The Cells gem comes with two helpers to invoke cells, both implement their own style and have a slightly differing naming convention.

Throughout this chapter, we used the `concept` helper which is made to render cells derived from - surprise! - `Cell::Concept`. Concept cells have a self-contained file structure and follow Trailblazer conventions.

1. You need to specify the full path to the cell class in the helper call, e.g. `concept("thing/cell")`, as the helper does not infer constants and does not know about any naming conventions.
2. Also, views in concept cells are structured the Trailblazer style. They sit in the concept's view directory. For instance, `app/concepts/thing/views/` will contain all views for a concept - unless you introduce sub-directories for a finer structure.

In contrast to this style is the `cell` helper. Here, the class has to be a `Cell::ViewModel` subclass. Usually, the class name is suffixed with `Cell` resulting in something as follows.

```
1 class ThingCell < Cell::ViewModel
```

1. The helper will automatically add the `Cell` suffix and allows to be invoked like `cell(:thing)`.
2. Views and the actual view model class are put in the application's cells directory. For instance, `app/cells/thing_cell.rb` would contain the class and views would sit in `app/cells/thing/`.

Personally, I am not using the `cell` style anymore as I find the Trailblazer concept-style better encapsulated. Also, the naming with concept cell is more appealing: `Thing::Cell` to me makes more sense than `ThingCell`. This book exclusively uses the Trailblazer style.

Summary

Encapsulating view code into view models is a good thing. Although I usually recommend people not to put everything into cells, I have never seen a cell that was "overkill". View code quickly grows and gets more complex. A cell class cleanly abstracts that and forces you to define interfaces.

In this chapter we implemented a neat view component to display a thing summary with link and other necessary data. By introducing a container cell, we hid data aggregation from the caller. This gives us an easily testable, rock-solid application component that brings reusability for free.

In later chapters we will revise the cells we just wrote and learn about caching, view inheritance, and polymorphic views.

The following chapter will go back to the domain layer. We're gonna build more operations, and forms with nested models.

Nested Forms

In this fifth chapter, I want to talk about Reform and the way it handles nested models. Forms can represent deeply nested object graphs. I want to create several, partly unrelated models via one grouping operation and via one form, and this is what we're going to learn in the next half hour.

Forms are an integral part of every user interface, user interfaces used by human beings. Visual forms are quite different to the way machine-operated APIs work. Rails tries to merge those two concerns in controllers, and we're going to see how Rails' "RESTfulness" breaks down as soon as you add supplementary usability to a page.

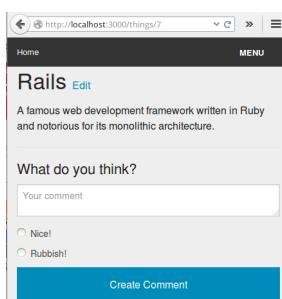
To demonstrate the misconception, this chapter will go the Rails Way and use nested resources following the limited CRUD mentality. Of course, this is not what I want, as this reduces the user experience to one form per page, with forms that solely represent database tables. This has nothing to do with *usability*. Nevertheless, this is what happens when you follow Rails' CRUD conventions.

In chapter 6, we restructure our UI and put the comment form where it should be in the first place: embedded in the thing page! Luckily, this is gonna be really simple as Trailblazer's design encapsulates workflows in operations and forms.

The code for this chapter is [available here²³](#).

Adding Comments

Our next milestone in Gemgem is to add *comments*. Users want to talk about things, so we'll focus on building the Comment concept in this chapter.



Following the Rails Way, given that we already had a Comment model, we start knocking together a CommentsController that allows creating or updating comments. This is a great convention. However, it is the opposite of user-friendliness.

The form to create comments will be on a separate page. That feels weird. I'd love to have the comment form right in the thing page, so when I visit, say things/1, I can instantly comment. Instead, creating comments goes onto a stand-alone page and the form is accessible via things/1/comments/new.

Rails requires you to nest "resources"²⁴ to define that the inner resource is dependent on the outer one. This is the case, as every comment needs a thing that it belongs to. Here's how config/routes.rb will get extended.

²³<https://github.com/apotonick/gemgem-trbrb/tree/chapter-05>

²⁴What they really mean is *models*.

```
1 resources :things do
2   resources :comments
3 end
```

At this point, I clearly have to state that I usually hate nested resources. They overcomplicate URLs, expose your database layout to your endpoints and create dependencies where they don't need to be. Wait for chapter 6, though, until we fix this.

When accessing the new comment form via `things/1/comments/new` this will be routed to the new action `CommentsController#create`. This is what's called "RESTful" in Rails. Let's see how the controller looks like.

The Controller

In order to render the form for comments, we will use our old friend `form` and a standard controller view for now.

```
1 class CommentsController < ApplicationController
2   def new
3     @thing = Thing.find(params[:thing_id]) # UI-specific logic!
4
5     form Comment::Create
6   end
```

By calling `form` I instruct Trailblazer to setup the `Comment::Create` operation and its form, both assets we're gonna implement in a minute (line 5). Recall that `form` will assign `@operation`, `@form` and the two identical `@model` and `@comment` variables. Yes, the model is aliased with a named instance variable. We'll make use of that in chapter 6.

Now, why am I putting logic into the controller and assign this `@thing` object (line 3)?

To compute the URL for our form, we need a reference to the outer "resource", too. One of the many problems nested resources create is *dependencies*. Even though this is purely UI-related logic, and we only need this instance variable in the view for URL generation, this feels wrong. Anyway, bear with it for this chapter, we will fix it very soon.



The Comment Concept

The next step we have to do is adding the comment concept in `app/concepts/comment`. As we know already, this is where all the code for this part of the application goes.

In the first draft of this new feature, a comment with text and a weight can be created. The weight qualifies the comment as either positive or negative feedback. This is a business requirement in Gemgem, as every input must be automatically interpretable.

Along with the comment we save an email address. At this early stage of the product, we don't have signed in users, but we already grab their email address.

Technically, I already know that we will have `users` at some point, so creating a comment implies the creation of an associated user.

Comment and User model

On the persistence layer, I add two new tables, `comments` and `users` along with matching models.

```

1 class Comment < ActiveRecord::Base
2   belongs_to :thing
3   belongs_to :user
4 end

```

Empty ActiveRecord classes make me happy. This model only contains associations, which is valid according to the Trailblazer specification.

It might help to see the generated schema for the `comments` table.

```

1 create_table "comments" |t|
2   t.text    "body"
3   t.integer "weight"
4   t.integer "thing_id"
5   t.integer "user_id"
6 end

```

Comments have a body which is the text, or the actual comment. They maintain a `weight` field and also have a `Thing` they belong to and are authored by one particular `User`.

```

1 class User < ActiveRecord::Base
2 end

```

The only relevant column in the User table is a text field named email.

This persistence configuration is sufficient code to get comments running in this chapter.

I am really happy that the ActiveRecord developers do such a good job and allow decoupling persistence and domain logic without major problems. And as we can see, it is really not that hard to maintain logicless models, once you make use of Trailblazer's additional abstraction layers.

Comment Operations

In order to create comments, nothing makes more sense than to start with the Comment::Create operation. The operation code will go into app/concepts/comment/operation.rb.

```

1  class Comment < ActiveRecord::Base
2    class Create < Trailblazer::Operation
3      include Model
4      model Comment, :create
5
6      def process(params)
7        validate(params[:comment]) do |f|
8          f.save
9        end
10     end
11
12   private
13   def setup_model!(params)
14     model.thing = Thing.find_by_id(params[:thing_id])
15   end

```

As always, the concept operations get nested into the model's namespace. By using the Model module and configuring it, the operation knows that it is supposed to create a brand-new Comment instance when its run (line 3-4).

The process method is almost identical to the one from the Thing operation. Nothing special is happening here, yet, unless you still find the validation of input followed by a model save fascinating. I do (line 6-10).

The `setup_model!` Hook

What might raise your awareness is the `setup_model!` method. We haven't met this helpful guy before. You might recall that a comment is always associated to a thing. That in turn means somewhere in the operation this relation must be specified.

The `setup_model!` method is the perfect place to prepare your model before it gets validated. This hook is run directly before process is invoked, but after the generic model was created, making it the ideal location to add or change attributes of the CRUD model. In case you're interested, check out `Operation#setup!` where all the prepping happens.

Since `model` is already available, I simply assign the comment's `thing` (line 14). The ID of the nesting thing is to be found in the parameters `:thing_id` field. This is an interface requirement to the operation user. It means the user has to provide that ID in any case, otherwise the operation will raise an exception that it couldn't find the `Thing` instance.

This is why I use the `thing=` setter. I could also use `thing_id=`, but this will make sure a valid thing ID comes in the `params` hash.

Be careful, though, with the `setup_model!` method when you write operations that handle both forms and documents like JSON. In the latter, the only available `params` key might be `:id`. If you rely on other data, make sure the controller provides necessary data for all formats in the `params` hash. Data from the incoming document is only available from the `validate` call onwards. But we'll talk about that in the API chapters.

As a side note, I started using bang-ed method names (e.g. `setup_model!`) for public hooks to signalize that their return value is irrelevant. This means you have access to a range of documented object properties, like `model`, but don't have to return anything.

Nested Contracts

Let's talk about the last bit of our new operation now: the contract.

```

1 contract do
2   property :body
3   property :weight
4   property :thing
5
6   validates :body, length: { in: 6..160 }
7   validates :weight, inclusion: { in: ["0", "1"] }
8   validates :thing, :user, presence: true
9
10  property :user do
11    property :email
12    validates :email, presence: true, email: true
13  end
14 end

```

Every comment has a text body, a weight, and a related thing (line 2-4). I add some boring constraints, for example, the comment body can only be 160 characters maximum (line 6-8). This is not a restriction, but a motivational factor as people don't feel urged to write a complete review.

An interesting validation is the presence requirement for `thing` and `user` (line 8). This validation doesn't know about associations, all it checks is if both fields are present - whether that is a boolean or a `Thing` instance the validation doesn't care. We will learn about high-level model validations later.

Nesting additional declarations into a property block creates a nested form. Reform now expects and handles nested input for the `user` sub form (line 10-13). In Reform, this is called *nested form* or inline form, as it is defined within an existing form class. Please note that you can define properties, validations and methods in that block. The block will simply be executed in a separate form class, making methods only available in the nested scope.

Nested forms are a great way to map multiple models to one visible form and run their validations together as a group. Even though the `Comment-->User` relationship here is a 1-to-1 mapping on the persistence layer, we will learn that a nested form is absolutely not limited to `belongs_to` and `has_many` relationships between models.

Instead, it doesn't even know about those implementation details. All a nested form cares about is composed objects. Those objects can be models, compositions, or separate objects as we'll see at the end of the next chapter.

Rendering the Comment Form

We have a controller action and an operation along with a form object in place. Now, let's see how the controller view rendering the form looks like.

To display the comment form I simply hack this form into the comments controller's view at `app/views/comments/new.html.haml`.

```
1 = simple_form_for [@thing, @form] do |f|
2   = f.input :body
3   = f.collection_radio_buttons :weight, [
4     [[1, 'Nice!'], [0, 'Rubbish!']], :first, :last
5
6   = f.fields_for :user do |u|
7     = u.input :email
8
9   = f.button :submit
```

In order to compute a URL for the nested comment, Rails requires two objects: the outer `thing` and the nested comment, which is represented by our form object (line 1). This is a clear drawback of nested URLs. The more you nest, the more objects you need for URL generation.

I then instruct `simple_form` to render a text area for the `body` property and render the two radio buttons for the `weight` in the clumsiest way possible (line 2-4).

The nested user form is best rendered using `fields_for` (line 6-7). All this helper does internally is calling `@form.user`, which returns the nested form object. It then does its convoluted magic to render the nested fields. I believe the `fields_for` helper implementation could be improved, but for now it helps, and that's why I use it here.

Browsing to a nested URL of an existing thing will show the form to add a comment. For example, if I browse to `things/1/comments/new`, an extremely inconvenient URL in my opinion, I can see the form to add a comment!

After the first boost of excitement has settled, we'll face a severe disappointment, though. While the comment form is visible, the nested user form is *not rendered*!

Pre-populating Forms

We have just encountered a feature in Reform that has caused a bunch of confusion in the community. Understanding why the nested user form is not rendered is crucial to understand what Reform really is: a form object, and nothing more. Now, why is the nested form not rendered?

The answer is very simple. The `Comment::Create` operation instantiates an empty `Comment` instance for us. We add the associated `Thing`, but we do not add or relate a user to it. This boils down to the following behavior.

```
1 comment = Comment.new # done in Operation::Model.  
2 comment.user #=> nil
```

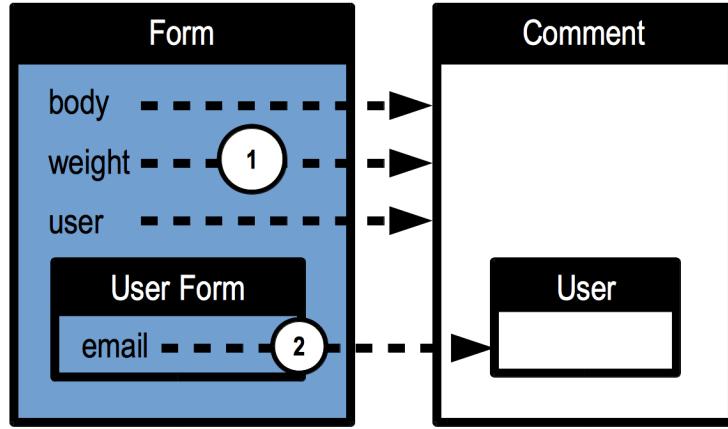
That is no magic at all. Not adding a user when creating the comment will result in an empty `user` property. I need to talk about how Reform populates itself in the initializer, now. After having discussed that, you will understand why there is no way for Reform to “guess” that it's supposed to display an empty, nested form for a non-existent object.

Reform's Population in Setup

Populating the form is run directly after the form got instantiated. As always, the initializer requires you to pass in a model that will be wrapped by the form. Creation of the form is hidden from you by the `Comment::Create` operation, however, it roughly works like the following code.

```
1 Comment::Create.contract_class.new(comment)
```

The main task of the form's initializer is populating the form with values from the wrapped model. This is called *setup population*, for the interested readers, this happens in `Reform::Form::Setup`.



Reform's population flow at setup.

Reform copies initial values from the model to the form. This is to create an indirection between the model and the form user, which can either be a reading form builder when rendering, a read-only validator, or a writing controller/operation when assigning incoming input to the form.

The whole purpose of the form object is to *not* let form builders, validators and writers access the model.

Copying attributes from the model happens by calling readers on the model. As you can see in the beautiful diagram, Reform simply invokes `comment.body`, `comment.weight`, and `comment.user` to do that (1). In a new form, this will normally return empty values, but it starts to makes sense in an editing form.

Usually, calling `comment.user` will return an object. In that case, Reform will automatically instantiate a sub form to represent that nested model. This composed form will then run its own setup workflow, again (2).

In our current environment, `comment.user` is empty. This is why Reform ignores this property: no nested form is built!

The missing detail now is the form builder. When calling `fields_for :user` it queries the form object for its user form. This does not exist, so the form builder will skip rendering that. And here we are again.

Setup Population via Reform

In order to make this nested form getting rendered we somehow need to create an associated `User` object. There are two ways to pre-populate the form. Let's explore the hard way first, as this is

an occasion to discuss some interesting features of Reform. I will then show you how to solve this particular situation with a simple one-liner.

Reform comes with an option `:prepopulator` that is designed to solve the above dilemma: pre-populate the form before it's rendered.

```
1 property :user, prepopulator: ->(*) { self.user = User.new } do
2   property :email
3 end
```

This creates a new `User` model and assigns it to the form. Using the form's `user=` writer makes sure a nested form is created, attached, and wraps the model accordingly. In a prepopulator, always make sure to use the form's public API to add models.

Note that the user is not associated to the comment object, per default. This is unnecessary for rendering.

When designing the pre-population hook in Reform I had to be careful. You can't simply run this automatically in the constructor of the form. This would work for form rendering, but blow up when *validating* incoming input. We will speak about this in a minute.

That being said, the pre-population logic has to be run manually before the form gets rendered. Here's a snippet to illustrate how that could look like.

```
1 form = Comment::Create.contract_class.new(comment)
2 form.user #=> nil
3
4 form.prepopulate!
5 form.user #=> <#UserForm wrapping new User model>
```

By invoking `prepopulate!` the defined callbacks are run and pre-populate the form. Luckily, you don't have to call `prepopulate!` manually in Trailblazer. This is implicitly run in the `Controller#form` helper, as this is the place where the form get rendered.

When browsing back to the new comment page, the nested user form gets rendered, and we're able to fill in an email address of the commentator! Hooray, we're ready to speak about the processing, now.

Form Processing

Per convention, when submitting the form this will POST parameters to `/things/1/comments` and this, in turn, gets routed to the `CommentsController#create` action.

```

1 def create
2   run Comment::Create
3 end

```

This action is far from finished, but it will create and run the `Comment::Create` operation, instantiate the form object and run the form's `validate` method passing in the input from the submission.

We'll refine the controller action pretty soon. However, this is all we need to run into the next problem nested forms yield. Well, let's not call it *problem*, it's more of a thing we have to understand in order to fully exploit the power of nested forms.

What did we get done so far? We have the comment form on its own page. When clicking "Submit", it gets send to a processing action `create`. That is pretty exciting.

Now, what happens when we click submit for real? Oh no! Another exception! This time, Reform complains that it doesn't know how to deserialize incoming data in `validate`. What is going on this time?

Populating Forms for Validation

When we run the `Comment::Create` operation in the `create` controller action, the internal form is instantiated, again. And again, we pass in an empty `Comment.new` instance which was created by `Model1`. So far, so good.

We already talked about the `run` helper, so you're aware that this will internally invoke the `validate` method on the operation's form while passing in the respective `params` hash. In the operation, this results in a call as follows²⁵.

```

1 contract = Comment::Create.contract_class.new(Comment.new)
2
3 contract.validate({
4   body: "Fantastic",
5   weight: "1",
6   user: {
7     email: "zavan@trb.org"
8   }
9 })

```

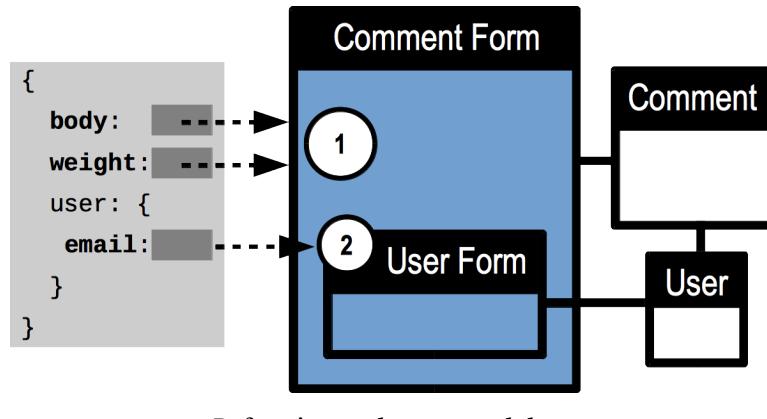
First, it creates the form object with an empty, new comment (line 1). The parameter hash in this snippet is not exactly what comes in, but represents it fairly enough. Nevertheless, these parameters are passed to the form's `validate` method, which will now try to assign the incoming data to the form graph (line 3).

²⁵When interested, check out the code in `Operation#validate`.

Deserialization in Reform

The following diagram tries to illustrate that form graph and what happens during deserialization in the `validate` method. On the left-hand side is the input that gets assigned to the forms in the middle. As you can see, models to the right are not involved in this process.

Every incoming fragment must have a corresponding form. These forms have to be set up, and this is what we call *validate population* in Reform.



It is important to understand that Reform writes incoming data to the forms, and not to the model. Have a look again at the illustration. When using `Form#validate`, data is assigned to the form objects. The wrapped models don't even know there's data incoming, they are treated as immutable at this point. Writing to models only happens in `sync` and `save`.

When the form assigns data, this works fine for `body` and `weight`, as this is simply written to the main form (1). Reform now sees input for the nested `User` form and tries to dispatch this, too, onto the nested form (2).

While this form does exist in the illustration, it does not in our case. There is no `User` form set up because the empty `Comment` does not have a corresponding `User`, yet.

Now, don't confuse this with the `:prepopulator` option we learned earlier. When validating a form, we don't call `prepopulate!` for reasons we will learn soon, and hence there simply is no nested `User` form to write data to.

Again, there is no way for Reform to know that you want exactly one particular `User` instance to be wrapped by a sub form. Reform tries to assign the incoming sub form content to a non-existent nested form, and fails!

The solution is to tell Reform that it should create a nested form for you when there's input for it. This happens via the `:populate_if_empty` option.

```

1 property :user, prepopulator: ->(*) { self.user = User.new },
2           populate_if_empty: ->(*) { User.new } do
3   property :email
4 end

```

With this addition, the validate call works.

When Reform hits the `user: { .. }` hash and finds there's no matching nested form, it creates a new `User` instance, builds a nested form that wraps the user, and then assigns, or deserializes the incoming data in the nested user form.

In `:populate_if_empty`, you simply return a model instance from the block. Reform will automatically wrap the model with a nested form for you and attach it to the parent form. This is because the `:populate_if_empty` option is run *per missing form*, Reform knows that it has to wrap the return value with a nested form.

After the population has happened, the actual validation is run to figure out whether all that incoming data is sane.

Prepopulation vs. Validation Population

I need to stress how important it is to have two different options for prepopulation and for population during validation. Although in our case it looks identical, this can be completely different code. We'll have several occasions where we need both options.

- Prepopulation using the `:prepopulator` option will blindly run the lambda and create nested forms around whatever you instantiated in the block. This assumes a blank slate, like an empty `Comment` object.

You're free to add additional logic into the `:prepopulator` block, e.g. to add another empty form to a collection that already contains three items. More on that in the next chapter.

- Validation population mostly happens via `:populate_if_empty`. There are more ways to populate but we'll learn about them later. This option is run inside the `Form#validate` method, before deserialisation and the actual validation happens.

The `:populate_if_empty` option has a different behavior: It knows about the incoming hash *and* about the existing model(s) in the nested form(s). The block is only run when there is an incoming hash, but no corresponding nested form.

As a sidenote, I'd love to say some words about Reform's architecture. Reform might look like a monster, with all those options and populator hooks and so on. Please don't mistake that as a monolithic setup.

Most of the hard data modelling and transformation work is done by the `Disposable` gem. The form is orchestrating between so called *twins* and Reform's API. `Disposable` handles renaming, compositions, population, and even things like hash columns. It is completely decoupled from Reform. I am a bit excited to talk about all that cool stuff in later chapters.

Rendering the form works. Submitting the form works. Processing and validating works. We can create comments. Yay!

While the operation and its form is fully functional, we need to add a bit of controller code to make the UI work. But let's get to that in a second because I want to complete the life-cycle of a form by discussing saving, first.

Saving Nested Objects

Forms can also instruct the persistence layer to save models. As you might have noticed already, the nested user model gets properly assigned to the comment after the Create operation has been run successfully.

This happens when calling `f.save` in the `Create#process` method.

```

1 class Create < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:comment]) do |f|
5       f.save # Form#save
6     end
7   end

```

I can't stress how important it is to understand how deserialization/population in Reform works.

Again, please recall that models are not touched until we save. That means, the instantiated `User` during the population is not associated to the `Comment` model, yet.

This is per design. Reform sets up a parallel object graph of nested forms to validate application state. Persistence is only involved as a last step - the result of a learning from mixing dynamic and persistent state as found in the Rails Way of working with models.

When calling `form.save`, what happens is that each form will push its properties onto its model. This is called *syncing*. The form will assign values using public writers, e.g. it will invoke `model.body=`, `model.weight=` and of course `model.user=`.

And this is how the user eventually gets assigned to the comment model. After syncing, every form invokes `model.save` on its wrapped model. This is when application state is persisted.

Note how Reform is database-agnostic. In fact, it only calls writer methods on the models and invokes a generic `save` method on it. Basically, this is what makes Reform work with any ORM you favor, or even with plain POROs.

Another thing to know is: calling `save` is completely optional. If your models don't need saving and only expect attributes to be set, you can use `Form#sync` as it is documented in Reform.²⁶

²⁶The two methods `sync` and `save` are implemented in the respective modules `Form::Sync` and `Form::Save` in Reform.

Flash messages and Redirecting

Before we head over to test the commenting let's write some more features. Cause I feel like it. I'd like to get redirected to the thing page after creating a comment, and a flash message saying "Comment created." or something would be cool.

Both redirecting and flash is controller affairs as both tasks involve knowledge about HTTP and sessions. And this is exactly the controller's job in Trailblazer.

```

1 def create
2   run Comment::Create do |op|
3     flash[:notice] = "Created comment for \#{op.thing.name}"
4
5   return redirect_to thing_path(op.thing)
6 end
7
8 @thing = Thing.find(params[:thing_id]) # UI-specific logic!
9 render :new
10 end

```

One cool thing about `run` is that it allows passing a block which is only executed when the operation's validation was valid and the process method run successful. In our case, that means, a comment was successfully added.

If this is what happened, I set a flash message to notify the world about this fabulous and successful creation (line 3). Note that I access the operation's `thing` method to grab the outer model - a method we yet have to implement.

After that, I instruct the controller to redirect to the `thing` page. This doesn't make sense right now, because we don't have that page, yet, but it will make sense very soon (line 5). The `return` statement is crucial, otherwise the rest of the action is executed, too, which must not be.

In case the validation failed, we re-render the `:show` view (line 9) to display the form with errors. This is why I copied the line to assign `@thing` to that action (line 8). This is not beautiful and a result of Rails and its static CRUD convention.

This controller only contains presentation logic. And that is a design concept of Trailblazer. All business logic is happening in the operation, the controller solely knows about which operation to use and that it may be successful or invalid.

Readers for Operations

To reduce knowledge about the operation, we expose a new reader method `Create#thing` to the controller. This could also be achieved via `op.model.thing` but this would reveal internals about the comment model - a detail I don't want to share with the controller.

We simply add a delegating reader to the operation. Tests will follow shortly.

```

1 class Create < Trailblazer::Operation
2   # ...
3   def thing
4     model.thing
5   end

```

It is a common practice to expose basic readers for an operation. However, don't confuse that with a decorator. When it comes to presentation logic, it is better to use a view model as we learned in chapter 3.

Static Form Population

I promised you to simplify the process of populating the comment form with the user. Using `:prepopulate` and `:populate_if_empty` is extremely helpful when having a dynamic setup, for example, when it is optional to nest a user into a comment. Chapter 7 will demonstrate dynamic forms.

In our setup, we *always* want a user nested in the form. This means we can statically add the user when presenting the empty form and when validating the input. You might have guessed it already: this is a job for `setup_model!` in the operation.

```

1 class Comment < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     # ...
4     property :user do
5       # ...
6     end
7
8   private
9   def setup_model!(params)
10    model.thing = Thing.find_by_id(params[:thing_id])
11    model.build_user
12  end
13 end

```

This looks way simpler.

Instead of advising Reform to prepopulate at creation and populate in `validate`, we do it ourselves in `setup_model!` (line 11). I invoke the comment's builder method which is provided by ActiveRecord and it totally does the trick.

Note that this already associates comment and user on the persistence layer, however, this is totally fine in this case.

Since we use static population, we don't need any population-related options anymore (line 4).

Static population is advisable only when nestings don't change. Later, we will have forms with a collection of nested forms where the amount of items changes dynamically. Obviously, static population won't work there.

Form Presentation Helpers

As a last improvement for this chapter let's touch up the form's view, just a lil' bit. Form and view don't have to be messy. Right now, I have a few things I don't like in the view.

```
1 = f.collection_radio_buttons :weight, |
2   [[1, 'Nice!'], [0, 'Rubbish!']], :first, :last
```

This is a very long line to render the two radio buttons for the weight. And, and that's why we're here, it creates redundant code. Have a look at the corresponding validation.

```
1 validates :weight, inclusion: { in: ["0", "1"] }
```

Even though this literally is just ones and zeros, this is overlapping code. I have to change both view and form code if I decide to represent *Rubbish!* with a -1 value instead of 0. We can simply move all code into the form class itself to keep knowledge in one enclosed place.

```
1 contract do
2   def self.weights
3     {"0" => "Nice!", "1" => "Rubbish!"}
4   end
5
6   def weights
7     [self.class.weights.to_a, :first, :last]
8   end
9
10  #..
11  validates :weight, inclusion: { in: weights.keys }
```

The class methods `::weights` contains the canonical value/label mapping (line 2-4). This can be used instantly in the corresponding validation where I call the method and extract the keys, from the hash, making it the original `["0", "1"]` array (line 11).

Since validations are defined on the class level, this all has to happen in the same context, the class. For the view, I define a second method `weights`, this time, an instance method. This method accesses the original `weights` hash and adds view-specific parameters (line 6-8).

Many people use Reform as a pseudo view model and this is totally fine. When replacing form views with Cells we will see what other neat tricks are there.

Anyway, check out the view now.

```
1 = f.collection_radio_buttons :weight, *@form.weights
```

Extremely readable and no redundant logic anymore. And the “helper” can simply be called on the `@form` instance - it’s just an instance method! By expanding the return value I can pass multiple values to the radio button generator.

Pre-selecting Values

As a last touch-up and to express my positive attitude towards life, I want to pre-select the “*Nice!*” button in the form. This avoids a validation error in case someone forgets to click one of the buttons.

If you’re thinking about using prepopulation to pre-select a value, you’re amazing. This is exactly how it’s done in Reform.

```
1 property :weight, prepopulator: lambda { |*| self.weight= "0" }
```

Again, I assign the desired value via the form’s `weight=` writer method. Whenever we call `prepopulate!` on the form this will set the `weight` property to a zero string which will result in the form renderer checking the “*Nice!*” button for us.

Be careful, though. In an edit form, you probably don’t want to statically select a value for a button but rather reflect the actual object’s state. We don’t have a comment edit form, yet, so this is cool. However, there a more flexible trick to default values of forms. We’ll discuss that in chapter 7.

Party on, Wayne! Everything is working! We can create comments! Hooray! Ah ah. Don’t start celebrating just yet. It’s time to write tests for the Create operation and its form code.

Operation and Form Tests

As per usual, I start with testing the operation code. Since I hate the current way of modelling our UI and maintaining a separate page to create comments, we're gonna write smoke tests for controllers in the next chapter.

And now let's be honest for a second. In retrospective, writing tests for application code has always made me feel awkward. *"Oh no, I have to write tests now. Where do I start? What do I test? Where will that test go? Will I test the model, or integration, or what?"* I still have this weird feeling when it comes to application tests.

Then I remember that I am not supposed to think when writing tests in Trailblazer.

By structuring the entire domain into operations with the same low-level API, it becomes pretty straight-forward to write first tests. I don't need to think about how to test this or that *functionality* - functionality is always represented as an operation with an extremely simple interface.

So I jump into `test/concepts/comment/operation_test.rb` and start writing a valid operation call. Life can be so simple.

```
1 class CommentOperationTest < MiniTest::Spec
2   let (:thing) { Thing::Create.(thing: {name: "Ruby"}).model }
3
4   describe "Create" do
5     it "persists valid" do
6       res, op = Comment::Create.run(
7         comment: {
8           body: "Fantastic!",
9           weight: "1",
10          user: { email: "jonny@trb.org" }
11        },
12        id: thing.id
13      )
14      comment = op.model
15
16      comment.persisted?.must_equal true
17      comment.body.must_equal "Fantastic!"
18      comment.weight.must_equal 1
19
20      comment.user.persisted?.must_equal true
21      comment.user.email.must_equal "jonny@trb.org"
22
23      op.thing.must_equal thing
24    end
```

```
25   end  
26 end
```

Comments are always associated to existing things, so I use the `Thing::Create` operation as a test factory (line 2).

Look at how I nest the `user:` parameters into the comment hash. This is because of the nested structure of the form (line 10). If you check out the parameter structure in a real HTTP request after submitting the form, you will see they got the same nesting.

One thing worth mentioning here is how I grab the user (line 20). I use `comment.user` which assures that the user object was really associated to the comment.

And notice that I also test the `thing` method, as this is public API of our operation and used in the controller (line 23).

Of course, I add several tests for checking the integrity of our validations. You can have a look yourself in the repo. Here's one example, though, to discuss how I often assert validations.

```
1 it "invalid email" do  
2   res, operation = Comment::Create.run(  
3     comment: {  
4       user: { email: "1337@" }  
5     }  
6   )  
7  
8   res.must_equal false  
9   operation.errors.messages[:user.email].must_equal ["is invalid"]  
10 end
```

First, I run the operation with invalid parameters (line 2-6), then I access the operation's errors object and test specific error messages. I use an equality test here as it's the simplest test possible.

You are now a master of nested forms. Mapping an arbitrary graph of models, or objects, to a form, encapsulating a group of validations in that class and consolidating further processing and saving of the models is an integral part of the Trailblazer architecture.

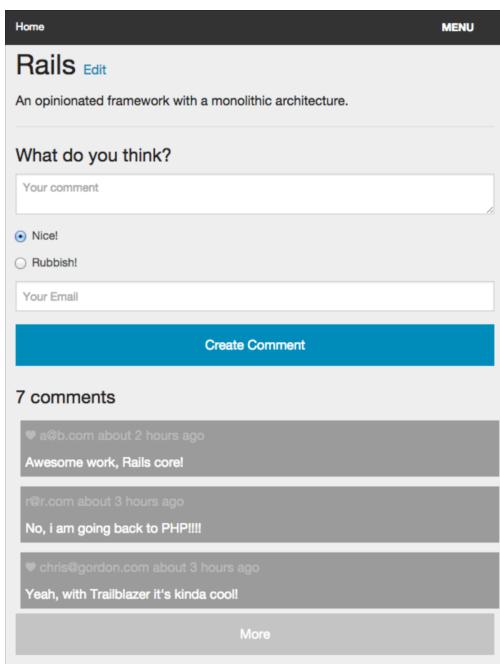
In Rails, this is usually done with a mix of preparing controller code, contextual validations, hackish validation code in models and, if it comes to the worst, `accepts_nested_attributes_for` to map validations and deserialisation semantics to nested model graphs. A shiver just ran down my spine.

Also, we've learned some more details about operations and how they are the ideal place for environmental setups and post-processing logic.

In the next chapter, we will rearrange our user interface and have composed views. Then, I'm gonna walk you through non-CRUD forms and operations.

Composed Views

In the last chapter we were being good, conservative people and obeyed the Rails Way. By embracing its CRUD, or “RESTful”, convention we quickly implemented the form to create comments. This ended up in a nested controller on a separate page.



thing we want to display.

```
1 class ThingsController < ApplicationController
2   def show
3     present Thing::Update
4   end
```

This code is sufficient to prepare the controller to display a thing. We still need to write the view, but let's talk about what happens in `present`, first.

The `present` Helper.

It must look a bit confusing to use the `Update` operation in a render-only action (line 3). Isn't an updating operation supposed to *change* application state? The answer is: Yes, it is. However, an

operation in Trailblazer has two faces. And this is not because they are suffering from bipolar disorder, but because of a reasonable object-design.

We already learned about a CRUD operation's main behavior where it is run and creates or updates one or multiple models. This was triggered using the `run` controller helper, remember that from chapter 3?

The present method represents the second semantic of an operation where only the setup code is run, without invoking the operation's `process` method. The `present` code is almost identical to `form`, which we used in chapter 3 to retrieve the form object of the operation.

Both `form` and `present` help you retrieving a particular model in the operation and then assign instance variables in the controller to access the operation, the form and the actual model in the view.

Let's recall the relevant `Thing::Update` operation parts, and then I'll walk you step-by-step through the flow.

```

1 class Thing < ActiveRecord::Base
2   class Update < Create
3     action :update
4     # ...
5   end
6 end

```

Here's what is going on when invoking `present` `Thing::Update` in the controller.

1. Since the operation includes the `CRUD` module and is configured to `find` (and `update`) an existing model, it knows how to retrieve the `Thing` model for the respective parameters. All that `present` does is telling `Update` to run its setup code using the `params` for the current request.
2. The `params` contain an ID for a particular thing, so the operation will find this model using your logic or the generic `CRUD` behavior.
3. Anyway, the operation does *not* run its `process` method, making it a read-only workflow²⁷.
4. After having run the operation's setup code, `present` assigns the aforementioned instance variables in the controller, like `@model`, and we're good to render the view.

Again, showing you the pseudo code that is run in Trailblazer itself might help understanding.

²⁷Internally, the operation only gets instantiated, nothing more. For a more functional design, though, I try to make the user not instantiate operations manually and therefore provide `present`.

```

1 def present(*) # receives operation class
2   op = Thing::Update.present(params)
3
4   @model = op.model
5 end

```

As you can see, the controller `present` helper really just forwards the `params` hash to the operation's same-named method, which will solely find the model and then return itself (line 2).

Reusing the `Update` operation is another great example how Trailblazer reduces redundancy in controllers. Usually, the `show` action would do that manually using `Thing.find` and assign instance variables.

This creates overlapping code that can become a problem. Say you change the way you retrieve things, you have to change that in every involved controller action. The fact that filters can help avoiding this is defeated as soon as it comes to access control and authorization.

Operations can also contain ACLs and more. It is impossible in a global controller filter to implement this level of granularity that we have by structuring our domain logic into operation classes.

Rendering a Thing

Fine, `present` retrieves the model using existing code and then assigns instance variables for the view. I guess it's time to have a look at `app/views/things/show.html.haml` now. I don't use Cells, yet. I want to use as much Rails Way as possible to make you feel the pain it will create, then we refactor the view to clean, fast view components in a later chapter²⁸.

```

1 %h1
2   #{@model.name}
3   = link_to "Edit", edit_thing_path(@model)
4
5 #{@model.description}

```

We simply render the thing's name, show a link to its edit form (line 3) and display the description so users get an idea what this is all about (line 5).

If you create a new thing discussion now, or browse to an existing one using a URL like `things/1`, you will see a minimalistic info page about the thing with title and description.

²⁸There is *Appendix A: Cells-Only Views* which completely replaces ActionView with cells.

Testing the rendering operation

One cool thing about reusing the Update operation for a render-only page is that we don't have to test it! Wheew, believe it or not - the invoked code is identical to the setup code when updating, and that is already covered in our operation test.

What we do have to test is the wiring in the controller action, plus its view. This goes into our smoke test in `test/integration/things_test.rb`, again. I simply add this after we created a new thing.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     # ...
4     click_button "Create Thing"
5
6     # show
7     page.current_path.must_equal thing_path(Thing.last)
8     page.body.must_match /Rails/

```

I can't mention it often enough how I love those smoke tests. Trailblazers test structuring has evolved over many years and makes it really easy to add a few lines of test that will increase sleep quality without pain.

And I know you already understood what we're doing in the test. After having submitted a valid form, we're getting redirected to the thing's show page (line 4 and 7). All I want to know is whether or not it renders the thing's title (line 8). A minimal test does what we need right now.

Now that we got the endpoint and view to check out things, why don't code the functionality to add comments to it?

Multiple Operations for Composed Pages

We want to render the comment form *in* the thing page. Plugging the form into `ThingsController#show` is relatively simple. We have to extend the old `show` action with a few lines of code.

```

1 class ThingsController < ApplicationController
2   def show
3     @thing_op = present Thing::Update
4     @thing    = @thing_op.model
5
6     form Comment::Create # overrides @model and @form!
7   end

```

Additionally to `present`, we now call `form` with our new operation, also (line 3 and 6). It is absolutely fine to run multiple operations in one action - as long as it is UI-specific!

I use the fact that `present` returns the operation instance (actually, all controller methods do that), and save it in `@thing_op` (line 4). To have access to the `Thing` model, that needs to be assigned to `@thing` manually (line 5). This step is necessary as the subsequent invocation of `form Comment::Create` will override the `@model` variable which we need in the view (line 6).

To get this straight: this is absolutely not the way I would build a user interface. Rails views, the lack of a rendering API and its instance variables are just not designed for composed views. Partials only hide the fact that we're relying on a lot of shared state.

Currently, the controller action is the place for composing the UI, so you might call more than one operation. However, never compose business logic in the controller. Again, this is purely for the user interface, and since we don't have a layer for composed UIs, yet, the controller is the right location for now.

After having collected data we can extend the view in `app/views/things/show.html.haml`. I am skipping the top part of the template file.

```
1 // ...
2 #@thing.description
3
4 = simple_form_for @form, url: create_comment_thing_path(@thing) do |f|
5   = f.input :body
6   = f.collection_radio_buttons :weight, *@form.weights
7
8   = f.fields_for :user do |u|
9     = u.input :email
10
11 = f.button :submit
```

It's important to understand that the `@form` instance variable comes from the `@form` operation invocation in the controller, so this is the `Comment::Create` form (line 4).

You will recognize the old `show` view appended with an additional form. Into the form-generating helper call I pass the `@form` object and a manual URL (line 5). That looks odd to an untrained, Rails-Way-spoiled eye.

Form Submission: Widgets vs. “RESTful”

Following the Rails Way, submitting the comment form goes to the `CommentsController#create` action. Per se, I find this a good and valid convention to structure controllers into CRUD actions, and resolve the selected action by considering the request type and the URL.

For example, in our case, a POST request to `comments/` would be dispatched to the controller's `create` action. In Rails, this is called "RESTful". I am still puzzled why this is named after the hypermedia-oriented architectural style, but let's call it quits for now.

However, this won't work for us. Remember, the comment form sits in the thing page. We could make that work for a successful submission, create the comment in the `CommentsController` and redirect back to the thing page.

What are we gonna do when things go wrong, though? I don't want the errors to show up on another page, I want the invalid form to be rendered exactly where it has been before the submission: embedded in the things page.

We've reached the limits of Rails' "RESTful" controller design.

It starts to break down as soon as you introduce a composed user interface. I simply don't favor a CRUD page to create a thing, and a separate page to create a comment, and so on.

I want to embed the comment form into the thing page. The comment form is more of a *widget*, a stand-alone view component that lives in the thing page without knowing about things at all.

So, how do we make this work? We need to validate the submitted form, create a comment on success or show errors on failure.

The answer from a Rails expert might be: "*Well, you are not RESTful, this is why you need an ugly work-around.*" and this is an extremely frustrating response, given that all I am trying is to improve usability. There is three things we can do now, according to the imaginary Rails expert.

1. I might get advised to use JavaScript validations. That way, we can use the "RESTful" approach, let the `CommentsController` create the *always valid* comment and redirect to `ThingsController` afterwards. This will always cause trouble since JavaScript can not do business-specific validations.
2. We can still POST to the comments controller, and in case of error, it redirects back to the original controller, passing a list of validation errors to display. This is extremely ugly and hard-wires error handling into the `ThingsController` that is absolutely not supposed to know anything about creating comments. I have seen that a lot of times in real projects. It made me cry. A bit.
3. AJAX! Rails' answer to fix the misleading "RESTful" architecture is AJAX. Send an asynchronous request to whatever controller and re-render the widget. I find this an OK solution when using real AJAX widgets as found in Apotomo and Cells. This is not the case in a vanilla Rails app and the amount of code I've encountered to implement this approach was depressing.

Luckily, there's another way to solve this.

Introducing a UI Action.

As you might have already guessed, we simply break the "RESTfulness" and introduce a new UI-specific action into `ThingsController`. I can hear Judas Priest playing *Breaking the Law* in the background.

This only sounds dirty. Since Trailblazer encapsulates all business logic into operations, the controller doesn't know anything about the form processing's internals. Let's look at the rendered form, again.

```
1 = simple_form_for @form, url: create_comment_thing_path(@thing) do |f|
```

To compute the URL for the form, I call a cryptic helper that is available because I changed the config.routes.rb file slightly.

```
1 resources :things do
2   member do
3     post :create_comment
4   end
5 end
```

Instead of trying to stay "RESTful" I simply add a new action (and a new route) to the things controller (line 3). This sends the submission back to ThingsController#create_comment where we process and re-render the page in case of an error.

This must feel dirty and wrong, but it does exactly what I want. Given that we will replace this kind of UI component with AJAX-backed widgets in later chapters, I can swallow the bitter taste and take it like a developer.

The new action, as already mentioned, goes into the ThingsController.

```
1 class ThingsController < ApplicationController
2   def create_comment
3     @thing_op = present Thing::Update
4     @thing    = @thing_op.model
5
6     run Comment::Create,
7       params: params.merge(thing_id: params[:id])
8
9     render :show
10  end
```

Let me start from the bottom, this time. The trick here is to render the old `:show` view, as we're actually coming from the `:show` action (line 8). In order to prepare all necessary instance variables and dependencies, we present the `Thing::Update` operation and assign variable exactly as we did it in the original action (line 3-4).

Instead of using the `form` helper, we now `run` the `Comment::Create` operation, which processes and validates the input, and then provides the form with potential errors in the `@form` object (line 6). Everything else is done in the view and the form builder.

The API of `Comment::Create` requires a `:thing_id` parameter. Since this invocation sits in `ThingsController`, that thing ID has to be remapped from `:id` to `:thing_id`, and that's what I do using the `:params` option for `run` (line 7).

And that's basically everything you need to build and process a composed page. I've seen this pattern a lot in vanilla Rails apps. However, those apps didn't have any encapsulation for business and processing logic. The result was chunky controller actions with tons of identical behavioral code.

We still have redundant code in our setup at the moment. I'm fine with that, though. The abstraction level of operations really boils down controllers to dispatchers that only know *what* to instantiate. They have zero knowledge about the internals and act as configured orchestrators, only.

Writing a Cells Feature

Now that we have the comment form in place and working by embedding it into the thing page, we're ready to add more UI functionality to that page. I am contemplating to render a list of recent comments under the comment form.

Once this works and we show the last ten comments, I want to allow the user to load more comments by clicking a "More" button. This button will load ten more comments in an AJAX call and append them to the existing list.

We will learn that this is incredibly simple with Cells and you will soon understand why I push you towards Cells' encapsulation when we implement the AJAX loader.

Why not start with the list of the last ten comments, embedded in that page? Using a cell to represent a comment, that could be rendered as follows.

```
1 concept "comment/cell", collection: @thing.comments.limit(10)
```

Naturally, I start with a comment cell in `app/concepts/comment/cell.rb` and the following code skeleton.

```
1 class Comment::Cell < Cell::Concept
2   property :created_at
3   property :body
4   property :user
5
6   def show
7     render
8   end
9 end
```

This is my usual workflow when writing a cell. I declare properties from the decorated model I am definitely gonna use in the cell, and add the `show` method.

Our next step is to write the view in `app/concepts/comments/views/show.haml`. This will represent the view fragment for one comment. However, as we want to display several comments, and at the same time make use of Foundation's grid, we need to add the respective CSS classes to each comment cell.

Remember from chapter 4 where we had the `Thing::Cell` that basically wrapped every thing into a markup container as follows?

```
1 <div class="large-4 columns">
2   <!-- cell content -->
3 </end>
```

In the comment cell, we need the exact same behavior: wrap the actual cell content in a container `div` and assign respective CSS classes to it. Let's extract that logic into a module that we can then use in `Thing::Cell` and `Comment::Cell`.

Here's the `show.haml` view of the comments cell.

```
1 = container do
2   - if nice?
3     <i class="fi-heart"></i>
4   = user.email
5   = link_to created_at, "#comment-#{model.id}"
6   = body
```

As you can see, the content is nested into a `container do ...` block (line 1). I will explain the rest of this view a bit later and want to focus on the container helper now.

Application-wide Features

Since we need this helper across different cells, I will write it in its own module.

In Trailblazer, application-wide features that can't be associated with one exclusive concept are put into `app/lib/:app_name`. For example, the new `GridCell` module goes to `app/lib/gemgem/cell-grid_cell.rb`.

The reason for `app/lib` is that this will allow reloading during development. Also, I've never understood the `app` directory located on the same level as the `lib` directory. To me, library code is to be shared within the application and hence should be in `app/`, also. If you want real libraries that are decoupled from any application, write a private gem.

```

1 module Gemgem::Cell
2   module GridCell
3     def self.included(base)
4       base.inheritable_attr :classes
5     end
6
7     def classes
8       classes = self.class.classes.clone
9       classes << "end" if options[:last] == model
10      classes
11    end
12
13    def container(&block)
14      content_tag(:div, class: classes, &block)
15    end
16  end
17 end

```

By putting the `GridCell` feature - and this module is a feature as it adds functionality - into the `Cell` namespace I visually categorize this module as a cells extension for my `Gemgem` application constant that is defined by Rails.

This module does three things.

First, I override the `included` class method (line 3-5). This is a common Ruby idiom to run additional code when this very module is included into a class (or another module). In our case, when `GridCell` is included into, say, the `Comment::Cell`, it will add an inheritable class attribute named `classes` to the latter. We can then use this to assign class attributes to our cell which we're actually gonna use for specifying CSS classes.

Maybe it's easier to understand when seeing how `GridCell` can be used to configure an actual cell. Here's the `Comment::Cell`, this time including the new feature.

```

1 class Comment::Cell < Cell::ViewModel
2   include Gemgem::Cell::GridCell
3   self.classes = ["comment", "large-4", "columns"]

```

The internal mechanics how that works are irrelevant right now. Fact is, we now can assign arbitrary data to the `classes` attribute.

When included, the `GridCell` feature also imports the `container` helper method (line 13-15) that we use in our view. This method is simple: it creates a `div` tag using Rails' `content_tag` helper, assigns CSS classes and then passes the block on to the helper (line 14). This will result in a properly configured `div` tag wrapping our actual view.

```

1 <div class="comment large-4 columns end">
2   Awesome work, Rails core!
3 </div>
```

By “properly configured” I mean that the wrapping div for one comment contains all the classes that we assigned - and more. Now, how do those classes get in there?

The container calls the instance method `classes` (line 14) which is defined in `GridCell`, too. This method does nothing more but accessing the class-wide configured `classes` (line 8) and then adding an `end` class to it in case the decorated model is the “last” one.

These are the same mechanics that we already discussed in chapter 4. By moving that into a module, we can reuse the “*let me define CSS classes per cell, render a container div and add an end class in case this very cell represents the last object!*” semantics for both things and comments.

I won’t discuss reusing the application-wide feature in `Thing::Cell` because you can [read the repository code²⁹](#) and have a look at this class yourself.

Simple Decorator Helpers

Beautiful, comments get rendered, wrapped accordingly in a container div, now let’s see what else is going on in a `Comment::Cell` view.

```

1 = container do
2   - if nice?
3     <i class="fi-heart"></i>
4   / ..
5   = link_to created_at, "#comment-#{model.id}"
```

One feature I implemented here is that positive comments are marked with a heart, cause, love is all around. This happens by putting a decider into the view that renders a Foundation heart icon (line 2-3).

Please note that I have absolutely no problem with simple deciders in views.

The `nice?` decider is implemented in the cell class. As usual, this goes into an instance method.

²⁹<https://github.com/apotonick/gemgem-trrb/tree/chapter-06/app/concepts/thing/cell.rb>

```

1 class Comment::Cell < Cell::Concept
2   # ...
3   def nice?
4     model.weight == 0
5   end

```

Now, be careful. This *is* business logic. Knowing that a zero represents a friendly comment is definitely not a thing that should be replicated in a view component. However, at the moment, the comment cell is the only place where we apply this wisdom.

We will extract that logic into a twin once we use it in other places, too.

Extracting The Timeago Helper

In the comment cell view, we also call `created_at` to display when that statement was exactly given. Again, I want Timeago to do its magic here and render a readable string instead of a raw timestamp.

We did that already in chapter 4. And, yeah, you guessed it: we're going to extract that into another cell feature in `app/lib/gemgem/cell/created_at.rb`.

```

1 module Gemgem::Cell
2   module CreatedAt
3     def self.included(base)
4       base.send :include, ActionView::Helpers::DateHelper
5       base.send :include, Rails::Timeago::Helper
6     end
7
8     private
9     def created_at
10      timeago_tag(super)
11    end
12  end
13 end

```

By overriding the `included` hook we can import necessary helpers into the cell that includes this feature (line 3-6). This means the cell does not have to do it anymore.

It simply includes this new feature and imports the `created_at` method that will use Timeago to render a readable timestamp (line 9-11).

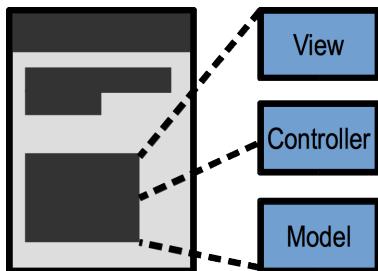
Rails' View Architecture

Look at the comment cell now, how brief the code is and how readable a view component can be.

```

1 class Comment::Cell < Cell::Concept
2   property :created_at
3   property :body
4   property :user
5
6   include Cell::GridCell
7   self.classes = ["comment", "large-4", "columns"]
8
9   include Cell::CreatedAt
10
11  def show
12    render
13  end
14
15  private
16    def nice?
17      model.weight == 0
18  end
19 end

```



Implementing components for a view fragment in Rails MVC.

At this point, I want you to think back of how that would work with partials, controller assigns and helpers. How would you dynamically configure a partial's CSS classes without passing around locals and using code in views?

It is impossible with Rails' standard view stack since it is missing the concept of *objects* that represent fragments, or widgets, in your user interface. The fact that partials internally use objects to render fragments is not what I'm talking about here. In Rails, the problem is that those objects are not part of the high-level architecture accessible for the application developer.

The view layer in Rails provides you with templates and functions. However, there's no direct relation between those two - it is incredibly hard to structure your view as you never know whether or not to do things in the controller, in the view itself, or in a dozen global helper functions.

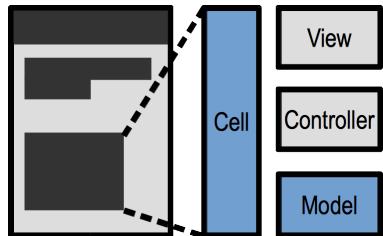
Cells, on the other side, simply represents a fragment of your UI with an object. This is applied OOP, nothing more.

A position shared by many people when they first hit Cells years ago was that Cells introduces “over-abstraction” to solve a “simple problem”. Instead of playing around with this new concept it was discarded because it “doesn't outweigh the increased technical complexity it brings”.

Look at the cell code we've written so far, and its view. Think about what it does. Think about the reusability like the wrapper container logic, the simple view and the few lines of code in a cleanly

separated class. I don't think this is over-abstraction at all, and I do not see where Cells brings "technical complexity". The opposite is the case: By applying encapsulation via a view component, I have one thing less to worry about.

Kaminari Pagination and Cells



View fragment implemented with a cell component.

Originally, we were rendering the list of comments manually in the thing's show view at `app/views/things/show.html.haml`.

```
1 concept "comment/cell", collection: @thing.comments.limit(10)
```

This works neatly in order to display comments. Nevertheless, we've got more things planned. We also want to allow users to load more comments by clicking a "More!" button, which involves incorporating a button into the view, and additional logic.

"*Wait, did you just say "additional logic" for the view? Let's put that in a cell!*" is what you were just thinking, weren't you? You're awesome! That's exactly what to do right now. Encapsulate the rendering of the comment list, or grid, dependent on our screen size, into a composing cell and let that embracing widget handle pagination. And AJAX. Of course.

To implement that grid cell, I push the code into `app/concepts/comment/cell.rb`. I am not gonna talk about the nesting-cells stuff, again, as this is already covered in chapter 4. Instead, let's focus on the "More!" button and the pagination logic.

But first, we need to define the new cell's interface. Here's how the grid cell is invoked.

```
1 = concept "comment/cell/grid", @thing
```

Note that I pass in a Thing instance, and not a list of comments. This is a dependency I create on purpose.

As the next step, we should have a look at the first implementation of the new cell.

```

1  class Comment::Cell < Cell::Concept
2  #
3  class Grid < Cell::Concept
4    include Kaminari::Cells
5
6    def show
7      concept("comment/cell", collection: comments) + paginate(comments)
8    end
9
10   private
11     def comments
12       @comments ||= model.comments.page(page).per(3)
13     end
14
15     def page
16       options[:page] or 1
17     end
18   end
19 end

```

The `Comment::Cell::Grid` cell is gonna sort out rendering a list and paginating it. I structure that into the existing cell, mainly to separate it from the global namespace.

In the `show` method I simply copied the original `concept` call for rendering the comments collection (line 7). I then append a generic pagination view. Of course, this is only prototyping, but I find it kinda cool how we can simply treat method results as strings and concatenate them to implement a quick UI draft.

Pagination with Kaminari

The `paginate` helper comes from the fantastic [Kaminari³⁰](#) gem which is well-established in the Rails community. Cells view models provide support for it using the `kaminari-cells` gem, a tiny compatibility layer that gets activated by including the respective module into the paging cell (line 4).

Kaminari gives us more helpers, but just to demonstrate how simple it is to use it, I use `paginate` now. This won't render a button, though, but a full-blown pagination row which we don't actually need. I'll fix this in a minute.

Let's see how Kaminari gets the data it has to paginate. The list of comments comes - surprisingly - from the `comments` invocation (line 7).

³⁰<https://github.com/amatsuda/kaminari>

```
1 paginate(comments)
```

This will call `Grid#comments` to collect comment records to paginate.

Aggregating Data in Cells

Of course, the `comments` method needs to be implemented in the `Grid` cell itself. Here's how that is done.

```
1 class Comment::Cell::Grid < Cell::Concept
2   # ...
3   def comments
4     @comments ||= model.comments.page(page).per(3)
5   end
6
7   def page
8     options[:page] or 1
9   end
10 end
```

In the `comments` method we do the actual pagination. Since the `Grid`'s model is a thing, not a comment list, we can call `model.comments.page ...` and run the pagination logic right in the cell, not in a controller (line 4).

Again, this is my decision as a software designer. Pagination is a visual usability concern and perfectly located in a cell. Once we need pagination elsewhere, we have to move it into an operation or a twin.

One important thing is that I memoize the comments list using the `@comments` instance variable (line 4). This is because this methods gets called multiple times in one rendering run. While I usually avoid memoization of values, this makes sense here as `comments` is a pure read-only query method.

Another detail is the `page` invocation which is another method in the grid cell (line 4). We simply provide a default page (which is the first page) in case the `:page` option wasn't passed into the cell invocation (line 7-9). This is the case when the comments get rendered initially. Remember how we invoke the entire comments grid?

```
1 = concept "comment/cell/grid", @thing
```

There's no `:page` option. We could pass this here by adding `page: params[:page]`, or whatever, but I found it interesting to talk about default values.

This code is enough to render the list of comments along with a pagination row. Clicking on a distinct page number will reload the page, but keeps rendering the first page. This is because we don't pass in the page number from the request as I briefly mentioned in the last paragraph.

Instead of the complete pagination element, I want to have one button saying "*More!*" that loads the next page via AJAX and appends it to the existing comments list.

AJAX Pagination with Cells

To add that "*More!*" button I've been talking about for 20 minutes now I introduce a new view for the grid cell. This goes into `app/concepts/comment/views/grid.haml`. Note that this is the same view directory that we use for the comment cell.

It is very common to put the same cell views into one directory, even though they are different classes. You can maintain separate directories for every cell class, and that is what happens per default. I want to show you how cells can share view directories, though.

Here's the `app/concepts/thing/views/grid.haml` view that will represent the entire list and pagination widget.

```

1 = concept( "comment/cell", collection: comments, last: comments.last)
2
3 - unless comments.last_page?
4   #next
5     = link_to 'More', |
6       next_comments_thing_path(thing_id: @thing, page: comments.next_page), |
7       remote: true

```

First, I render the comments collection using the `Comment::Cell`. I also pass in the `:last` option so the last cell knows it has to add the `.end` class to its container div (line 1).

Unless it's the last page, I then render the notorious "*More!*" button.

Note that `last_page?` is a Kaminari method on the comments collection. This will suppress rendering of the button on the last page of comments (line 3).

The button gets nested in a `#next` div (line 4). A clumsy URL helper then provides the path where to send the AJAX request (line 5). In this URL needs to be embedded the thing id and the page number of the next set of comments. Again, I use a Kaminari method called `next_page` here.

By specifying `remote: true` I advise the link helper to create an AJAX link. When clicking, an AJAX request gets sent to a path like `/things/3/next_comments?page=2`.

Rendering this `grid.haml` view has to be done explicitly. Here's the show state of the grid cell.

```
1 class Comment::Cell::Grid < Cell::Concept
2   inherit_views Comment::Cell
3   # ...
4   def show
5     render :grid
6   end
7 end
```

As you can see, you can pass the view name to `render` (line 5). When skipping the view name, Cells will try to render `show.haml`, which is the wrong view.

Also, you have to configure the cell to share the view directory with the comment cell. This is done by using `inherit_views` which will basically instruct Cells to look into `Comment::Cell`'s view directory when rendering a view (line 2).

Instead of searching for `comment/grid/views/grid.haml`, the cell will now also check for `comment/views/grid.haml` which is exactly where we put the view.

Sharing view directories is not necessarily enforced by Trailblazer. I just find it convenient to have one view directory per concept, unless I want to write really reusable cells.

AJAX Processing Cells

Of course, this special URL needs to be intercepted by a controller action. In Rails, we have to add a route for this. The following addition to `config/routes.rb` implements this additional UI-specific route.

```
1 resources :things do
2   member do
3     post :create_comment
4     get :next_comments
5   end
6 end
```

Instead of trying to stay “RESTful” I add another route to the `things` resource routes, namely the `next_comments` action (line 4).

This will be routed straight to `ThingsController#next_comments`. Why not go and check out this new action.

```

1 class ThingsController < ApplicationController
2   # ...
3   def next_comments
4     present Thing::Update
5
6     render js:
7       concept("comment/cell/grid", @model, page: params[:page]).(:append)
8   end
9 end

```

And this is the point where you will realize what *reusability* really means. Reusability does not only allow sharing behavior between projects, it also, and that's the point about it, helps taking advantage of a component within one and the same application!

That is exactly what we do here. To render the next page of comments, we make use of the comment's grid cell.

Remember, the only dependency for the grid cell is a `Thing` instance. As we learned earlier, the `present` invocation will prepare the controller by grabbing the `thing` instance that is requested as per URL (line 4).

After setup the action, I call the grid cell, pass in the thing and the page number from the request (line 7). The page number's the optional second dependency for the our cell and will result in the cell rendering the correct page.

One thing to notice is that I instruct the Rails controller to return the rendered fragment as JS (line 6) which makes it work seamless with Rails UJS layer.

Call Styles in Cells

Please, listen up now. This time when invoking the cell, I do *not* call the `show` method. Instead of letting the `concept` helper trigger the default `show` cell method, I do it manually.

Here, I invoke the `append` state using the *call style*, as we say in Cells (line 7). The syntax used here is the preferred way to call an alternative state when invoking a cell: `cell.(:state)` is an alias to `cell.call(:state)` but looks more nerdy and underlines the limited public API that I'm trying to enforce with all my gems.

You might wonder now how many different ways there are to invoke methods on cells, and the answer is: Two. Only two.

1. You can always call a method directly. A cell is an object and objects respond to methods.

```
1   concept("comment/cell/grid").append #=> <div ...>
```

This will really just call the `append` method (or, `show`, or whatever you want) and provide the method's return value. However, this neither does involve caching, nor does it mark the string as `html_safe`, a horrible construct found in Rails to avoid automatic HTML escaping of strings.

2. The preferred way, nevertheless, is the *call style*. And this is what the `concept` helper does for you in case you didn't call a method explicitly. The invoked method defaults to `show`, but since we want to call `append` I use the call style to achieve that.

```
1 concept("comment/cell/grid").(:append) #=> <div ...>
```

The call style will mark the returned string as "safe" and, more important, involve Cells' caching³¹. We'll talk about caching soon.

My advise is to use call style when you need to trigger an alternative method.

On Controller Structuring

The newly added `next_comments` method is another aggravating break with Rails and its "RESTful" controller paradigm.

It becomes obvious that structuring actions in controllers gets degraded, controllers become stupid endpoints that dispatch to operations or cells.

Your software architecture is no longer focused on which controller is "RESTful" and which is UI. When you need a new UI function, you chuck an action into the closest controller to back that function's request. The point here is that all our logic is encapsulated in appropriate objects. Those operations or forms do not care whether they're run from `CommentsController` or `ThingsController`.

Controllers become lean HTTP endpoint dispatchers, and that is a good thing.

Appending per Javascript

We click the "More!" button. An AJAX request triggers `next_comments` in the things controller. This in turn invokes the grid cell and calls the `append` method. The result is then sent back to the browser which runs whatever was the result of `append`.

So, the last missing piece is `Grid#append` and here we go.

³¹Reading the `call` code definitely helps. You'll see that the implementation is simple and straight-forward. Go check out the `Cell::ViewModel#call` method to see how it's done!

```

1 class Comment::Cell::Grid < Cell::Concept
2   include ActionView::Helpers::JavaScriptHelper
3
4   # ...
5   def append
6     %{ $("#next").replaceWith("#{j(show)}") }
7   end

```

The `append` method does nothing more but calling the original `show` method which will render a HTML fragment with the next page of comments. I escape the content using the `j` helper from Rails' `JavaScriptHelper` (line 6). This helper has to be included into the cell in order to be available (line 2).

This string is then simply wrapped by the appropriate JavaScript code to append the new page of comments to the list found on the page. I do this by replacing the old `#next` div with the new content. Its advantage is that this removes the old “More!” button while adding the new comments.

I'm sure there's better ways on the JavaScript end to achieve that, but it basically does the trick.

Cells Tests

Postponing tests in this chapter has been part of my agenda. Until we have a working view component in place it doesn't make sense to frantically follow the TDD approach just to say you're frantically following the TDD approach.

We could have started with a cell test for the `Comment::Cell` but I prefer to test the entire grid component as one. Since we do not use the single comment cell anywhere it doesn't make sense to test this in isolation. Testing the latter would mean we test a private concept, which isn't helpful when refactoring.

Now that everything is working, it definitely is time for tests. And, hey! Tests with Trailblazer are fun! Keep telling that to yourself, it works.

Let's quickly run through some of the tests for the comment cells. As aforementioned, I only test the `Comment::Cell::Grid` cell. Since I'm not gonna discuss every single line you should have a look at the [test file³²](#) for a complete picture.

I start with a test setup. Being a crazy Trailblazer, operations are used for this.

³²https://github.com/apotonick/gemgem-trrb/tree/chapter-06/test/concepts/comment/cell_test.rb

```

1 class CommentCellTest < Cell::TestCase
2   controller ThingsController
3
4   let (:thing) { Thing::Create.(thing: {name: "Rails"}).model }
5
6   before do
7     Comment::Create.(comment: {body: "Excellent", ..., thing_id: thing.id})
8     Comment::Create.(comment: {body: "Well.", ..., thing_id: thing.id})
9     # two more...
10  end
11 end

```

Of course, we need a thing to present, and this is what I do first (line 4). After that, I create four comments. Since one page contains three comments, we will have to paginate once, which is exactly why I chose this number.

Note how I use the earlier created `thing` to associate the comments to the thing (line 7 and 8).

```

1 it do
2   html = concept("comment/cell/grid", thing).(:show)
3
4   comments = html.all(:css, ".comment")
5   comments.size.must_equal 3
6   #..
7 end

```

Following the setup code we can start testing - yay! I grab the markup string by invoking the grid cell and its `show`. Since this requires a model I pass in the fixture `thing` (line 2).

The returned string is ready to be tested with Capybara matchers and this makes it incredibly simple to assert markup constraints. First, I test if we really display three comments by checking whether we have three `.comment` divs (line 4-5).

```

1 it do
2   # ..
3   first = comments[0]
4   first.find(".header").must_have_content "hilz@trb.org"
5   first.must_have_content "Improving"
6   first.wont_have_selector(".fi-heart")
7   first[:class].wont_match /\send/
8   # ..
9 end

```

I then grab the first rendered comment which represents the most recently created (line 3) and test whether the right author and comment body are in place (line 4-5). Since this comment is a negative one, an important test is to check that the heart icon is not rendered (line 6). As the first comment in the list, it shouldn't have the `.end` class, this is for the last comment (line 7).

The same thing happens for the second comment. This one is a positive comment. Generally, I make sure to test both positive and negative comments and always make sure the `.end` class is not assigned.

Here's an excerpt from the third and last comment in that current list.

```

1 it do
2   #
3   third = comments[2]
4   third[:class].must_match /\send/ # last grid item.
5   #
6 end

```

Here, the wrapping div must contain the `.end` class (line 4).

Note how simple Capybara makes it to assert CSS selectors on fragments in your view. You can use finders and then apply selectors to parts of the view by using well-known Ruby operations like `#[]`.

As a last test, I check that the “More!” button gets rendered.

```

1 it do
2   #
3   html.find("#next a")["href"].
4     must_equal "/things/#{thing.id}/next_comments?page=2"
5 end

```

This checks two things. First, I make sure we got a `#next` div as this is needed for the AJAX update. Second, I check that the URL of the wrapped link is correctly pointing to our AJAX action with the right parameters (line 3-4).

I don't use URL helpers to generate the target address in the test. For some strange reasons, I feel safer to assert a “real”, hardcoded URL instead of letting the test generate it. This test will break, of course, when I change my routing. As routing changes rarely happen, I am happy with this test.

Testing Pagination

The pagination action append in the grid cell wraps the identical code into a JavaScript statement. I am not gonna test everything again here but write a super minimal test for now.

```

1 it do
2   html = concept("comment/cell/grid", thing, page: 2).(:append)
3
4   html.must_match /replaceWith/
5   html.must_match /zavan@trb.org/
6 end

```

This time, I invoke the `append` method and pass in the next page (line 2).

All I assert is that the JavaScript is rendered (line 4) and if the fourth comment is in that page (line 5). In my lousy test I simply check if the fourth commment's author email is in the rendered markup.

When I said lously I meant lousy. Here, I expose knowledge about the implementation in my test. Since I know that the `append` method calls `show`, I skip testing details and only test the added code on top of that.

However, this is way better than having no test at all. In fact, I test everything required to assure the component works the way we expect it to operate. Both JavaScript generation and the actual pagination process are tested in a, well, not too obvious way.

Let us come back to more mature tests at a later point.

Smoke Tests

As a last step we need to write some more smoke tests for the new actions in `ThingsController`. First, we test the rendering of the new comment elements in the `thing`'s `show` view. This goes into the `test/integration/thing_test.rb` test.

I copied the `thing` fixture creation from the `cell` test. I did this on purpose - we will soon replace redundant test setups with factories. Factories that use operations, of course.

We have tested the `show` action already, but now we need to add tests for the embedded comment form plus pagination. Here's the extended test case for it.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     # ...
4     page.body.must_match /Rails/
5     # comment form in show
6     page.must_have_css "input.button[value='Create Comment']"
7     page.must_have_css ".comment_user_email"
8     page.must_have_css ".comments" # grid.
9   end

```

I insert this test code into the `#show` section of our integration test, and check whether the comment form is displayed (line 6). Just to make sure the input fields are rendered, I check for the nested email field (line 7).

Since we also show the comment grid I try to find the comments, or the grid. This assures that the `Comment::Cell::Grid` component is rendered (line 8).

Create Comment Tests

Tests for creating comment and pagination are integration tests as well, and will get their own test file `test/integration/comment_test.rb`.

When integration testing, I tend to structure by concept, too. While we put all thing-relevant scenarios into `thing_test.rb`, this is now very specific to comments, the fact that the code is implemented in the `ThingController` to me is irrelevant here.

Again, I copied factory code for now and will skip it here.

```
1 class CommentIntegrationTest < Trailblazer::Test::Integration
2   describe "#create_comment" do
3     it "works" do
4       visit "/things/#{thing.id}"
5       fill_in "Your comment", with: "That green jacket!"
6       choose "Nice!"
7       fill_in "Your email", with: "seuros@trb.to"
8       click_button "Create Comment"
9
10      page.current_path.must_equal "/things/#{thing.id}"
11      page.must_have_css ".alert-box", text: "Created comment for \"Rails\""
12    end
13  end
```

To create a comment, I visit the thing show page, fill out the form and submit (line 4-8).

After successful creation, the controller must redirect to the new thing URL (line 10). I also test whether the flash message is correct (line 11). The flash message is a UI-specific feature and both implementation and test go into the controller layer.

In the repository test file, you will find an additional test for an invalid submission. Again, this purely tests the mechanics for now and no details or error messages are tested.

Pagination Tests

Last but not least, and it has been a long way, I test the pagination of comments, implemented in `next_comments`. Being extremely exhausted from all those tests and all that smoke, this goes into a real simple one.

```
1 class CommentIntegrationTest < Trailblazer::Test::Integration
2   describe "#next_comments" do
3     it do
4       visit thing_path(thing.id)
5       click_link "More!"
6       page.must_have_content /zavan@trb.org/
7     end
8   end
9 end
```

I visit the show page and click the pagination link “More!” (line 5).

After the request I simply check that the last comment is somewhere in the response’s body (line 6). This is similar to the cell pagination test we wrote earlier. Even though this test looks as if it doesn’t really test anything, it will find and report if we ever break the pagination wiring in the controller.

Summary

Some cool stuff has been going on in this chapter. We’ve introduced new UI components to enhance our user experience. Comments are now directly embedded in a thing page via a presenting cell. And, even better, a form now allows to create comments right in that very same page.

Even though we broke with Rails conventions a lot here, the code remains clean and readable. Specific logic is hidden in operations and cells, making it unnecessary to worry about controller structuring any longer.

This is why both comment creation and pagination are happening in the `ThingsController`. And does that hurt? No, it doesn’t!

The next chapter will discuss forms a bit more, as they are the pivotal element in every application. Luckily, Trailblazer makes it relatively simple and you will be mastering your forms soon enough.

Mastering Forms

Forms. For some people, that is just a collection of input fields, buttons and checkboxes. After a few cumbersome seconds of filling out, you punch the submit button and wait. However, forms are more. Forms are the interface between users who are mostly human beings, and the machine. A form is the only way to communicate with the computer.

I know, this sounds dramatically, and there should be the theme of *Terminator II* playing in the background now.

But think about it. Operating a machine as an end user and telling it what to do needs an interface. And it doesn't matter whether this interface is a command line, a document-based JSON API or a beautiful web form with colorful sliders and a grid-backed layout - once the input got submitted, it's needs to be validated, processed and further actions have to be taken.

This is the reason why I decicate this chapter to complex forms, again. Forms will always be the pivotal element of web applications. We will learn how to dynamically populate an object graph, validate it and persist the graph to the database.

In the next chapter, I will discuss callbacks and how they can be attached to hooks in the form to post-process the application state change.

The code for this chapter is [on Github](#).³³.

Things and Authors

Do you remember when we started working on creating *things* in chapter 3? We implemented a form with title and description for a thing.

A screenshot of a web form titled "ActionView". The form has a text input field with the placeholder "An outdated template engine.". Below this is a section titled "Do you know any authors?" with three email input fields. The first field contains "none@knows.com" with a "Remove" link. The other two fields are empty. At the bottom right is a blue "Update Thing" button.

A few dozens pages later we wrote the functionality to add comments to things. In the comment form you did have to specify an *author*, or better, their email address.

I would like to extend the thing form now.

Say you add your own gem as a thing to Gemgem, or a book you know the author of. It'd be cool if you could add authors just as we create or update a thing.

Authors of things can either be existing or will be created on the fly. In the following chapter, we will then use callbacks to notify users of their new authorship, to join Gemgem, and so on.

³³<https://github.com/apotonick/gemgem-trbrb/tree/chapter-07>

Adding Authors

Regardless of how we visually turn over that feature, the data modelling is quite obvious. Things need to be associated with users. Luckily, we already added the `users` table and model in chapter 5.

Being good Trailblazers, we're allowed to add associations to the ActiveRecord model class. That's where ActiveRecord is really good at. Its relational table abstraction is amazing, too bad it got stuffed with unnecessary, confusing features like callbacks and observers.

Anyway, while I wrote some migration code to extend our database schema, the only important addition here is that after this change the `Thing` model knows about the relationship to authors, or users.

```
1 class Thing < ActiveRecord::Base
2   has_and_belongs_to_many :users
```

In the final version of the new thing form there will be three author fields where emails can be filled in. I simply decided that three is a nice threshold to begin with. Allow people to add a maximum of three authors. We can always increment this limit later.

I'm a very visual person. Actually I'm not. But let's pretend I am. Why not start with making the actual web form display three user email fields under the existing elements, just as you can see it on the screenshot above.

This starts with adding a new property to the `Thing::Create` contract.

```
1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3
4     contract do
5       # ...
6       collection :users
7       property :email
8       validates :email, presence: true, email: true
9     end
10    end
```

I use `collection` to declare that this form is now supposed to take care of a bunch of authors. Note that I call the property `users`, even though we speak of authors (line 6). We could rename the association in `Thing` and all that, but at the moment, there's no need to do that.

Since we want to display an email field, I add just that, along with a few validations. The email always has to be filled out, and it has to be a valid email (line 8).

Dynamic Prepopulation

This form is rendered in `ThingsController#new`. When browsing to `/things/new`, we get disappointed, again. The form builder didn't render any user or author fields!

And I am pretty sure you know the answer, yet. There are no users attached to the brand-new Thing instance that was created when presenting the `Thing::Create` operation. We could override the `Operation#setup_model!` method as we did it in chapter 5 and add users there.

Nevertheless, I'm more and more becoming a fan of Reform's prepopulation semantic. It's an easy and clean way to add models to the form so they get rendered when presenting the form. In order to do so, all we have to do is adding a `:prepopulator` option.

```

1   collection :users,
2     prepopulator: :prepopulate_users! do
3       property :email
4       validates :email, presence: true, email: true
5     end
6
7   private
8   def prepopulate_users!(options)
9     (3 - users.size).times { users << User.new }
10  end

```

Now this looks a bit different to the prepopulator we wrote in chapter 5. I did this on purpose to show you two alternatives of implementing dynamic parts of the form. We were using a lambda block right in the property definition in the earlier chapter. This is cool for short assignments.

In this form, I use an instance method to implement a prepopulator. Reform allows that: when you provide a `:symbol` it will call just that method when the option is invoked (line 2 and 8-10).

What I do in the new prepopulator are two things.

First, I find out how many empty, or new, users to add to the form. This works by querying for the actual user count in the form (`users.size`) and subtracting that from 3 (line 9). In our Create use case, the initial size will always be zero, as no user is added, yet, but it starts to make sense in the Update version of this form.

After finding out how many, I add the new user objects to the form. By using the form's `#users<<` writer (line 9) nested forms can be added. Please *always* use the form's public API when adding, removing or changing nested forms. I keep stressing this because `#users<<` doesn't simply add the user instances, it wraps them properly in nested user forms and attaches them to the main form.

Keep in mind that your form object graph is a mirrored version of your actual model graph. Assuming that the thing model keeps three users (or authors) attached to itself, the thing form will contain three author forms, that in turn wrap one user model each.

This allows you to add, change and remove nested forms from the parent form without propagating that to the persistence layer. In our example, we add users to the thing. However, and now listen, this does not happen on the model layer, but on the form, only. The underlying models do not know about changes of the form graph, yet.

The separation of domain and persistence is a crucial point in Reform. You can play with your domain object graph as much as you want, add forms, change values, and so on - it won't touch the database before you hit `#sync`. We'll explore population a bit more in a few minutes. In a later chapter, when we talk about the *Twin* pattern, we'll go into detail how this is possible.

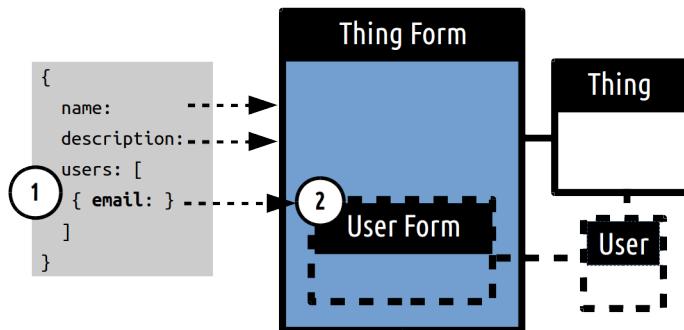
When we browse to the `/things/new` page now, the form is displayed and offers us to fill out three author email fields. Yay!

Validation Population

As soon as you fill in at least one email address and submit the form, we get a familiar exception.

```
1 RuntimeError:
2 [Reform] Your :populator did not return a Reform::Form instance for `users`.
```

The form couldn't be validated because we didn't populate it properly. This is something we already know from chapter 5. While we managed to get the rendering of the form running, we missed to write a populator for validation.



The parent form is missing nested user forms to process the input.

Here's the steps explained once again.

The incoming form input contains an array, each item fragment represents the input for one nested user form (1). Reform tries to assign each user hash to a nested form, but there is no nested form, yet (2). Please keep in mind that we did not call `prepopulate!` as this is only meant for presentation, not for validation of the form.

In order to dynamically create nested user forms, we need to take care of validation population. How tiresome, this whole form processing thing!

```

1 collection :users,
2   prepopulator:      :prepopulate_users!,
3   populate_if_empty: :populate_users! do

```

This is where `:populate_if_empty` enters the stage, our old friend from chapter 5. This option is only invoked when the incoming hash fragment doesn't have a corresponding nested form to be assigned to.

Again, I use an instance method to implement the option as this feels less clumsy (line 3). The referenced method has to be placed in the contract that defines the option.

```

1 contract do
2   # ...
3   def populate_users!(options)
4     User.find_by_email(options[:fragment]["email"]) or User.new
5 end

```

In the populator method, we have access to the corresponding runtime options. Most commonly, you will only need to access the `:fragment` part, which is the incoming fragment in `params` that is mapped to this nested form (line 3)³⁴.

This method is called for every hash fragment that does not have a nested form, yet. As per documentation, the return value of an `:populate_if_empty` option will be automatically wrapped in a nested form and added to the main form.

Pretty convenient, this option, so all I have to do is try and find an existing user with the provided email address or create a new user object (line 4). If I were an ActiveRecord expert I'd use `find_or_instantiate`, but I'm not, so I do it manually.

When filling in an email address from an existing user, our populator code finds that user, and Reform wraps it in a nested form. If the email is not existing, yet, a new user object is created and gets wrapped by Reform, too.

At the same time, when filling in some bogus email addresses in the provided fields and submitting the form, we get an erroring form complaining about invalid email addresses. So, validation seems to be working now. Strike!

Skipping Blanks

One thing that could be improved is unnecessary validation errors: Currently, we get errors for invalid emails even when the email fields are blank.

³⁴In case you want to learn more about the other available parameters make sure to check the [API documentation](#).

The reason is that Reform doesn't know that a blank string is not an email, so when populating the form in `validate`, it creates a nested user form for blank emails, too. While this is the correct behavior in our case this is not desired.

Luckily, Reform comes with an option to do just that: `:skip_if`.

```
1 collection :users,
2   prepopulator:      :prepopulate_users!,
3   populate_if_empty: :populate_users!,
4   skip_if:           :all_blank  do
```

The `:skip_if` option allows to dynamically decide whether or not an incoming fragment should be deserialized to populate the form. The shortcut `:all_blank` is a macro that will skip population when all fields of the form are empty. Adding this option will prevent Reform from adding empty users (line 4).

In a later section we're going to discuss how to use the `:skip_if` option with a hand-written piece of code. This option is extremely helpful for various reasons - you'll see.

We can create things, add existing and new users and skip empty forms automatically. Pretty cool. I don't know about you but I'm ready for the next business requirement!

Form Debugging Techniques

Form population is something you might not be used to from Rails, which does population "somehow".

I want to briefly talk about what has happened right now. To me, it's extremely important that future Trailblazers understand the concepts and thoughts behind the form object graph representing your domain, and the decoupling of the persistence and models.

An easy way to find out what the form really did in `validate` is to introspect it in the operation. In the next few sections we're gonna play in `Thing::Create#process` to learn more about form internals.

Usually, I use `raise` and `inspect` to find out what is going on at a particular point in my application. Surely, a debugger tool would help here, but let's do it the simple way for now.

```

1 class Create < Trailblazer::Operation
2   #
3   def process(params)
4     validate(params[:thing]) do |f|
5       f.save
6     end
7
8     raise contract.inspect
9   end

```

A simple `raise contract.inspect` will print you the - horrible looking - form instance right into your browser. As I put the `raise` statement after the `validate` block, I get insight into the form after validation has happened (line 8).

For example, say you enter an invalid email address. You could then change the `raise` statement as follows to inspect the actual errors.

```

1 raise.contract.errors.to_s
2 #=> {"users.email"=>["is invalid"], :name=>["can't be blank"]}

```

A quite convenient way to see what the validations really assert. Instead of trusting the form builder and its rendering of error, this is a quick way to get to the bottom of the internal workflows.

Now, to understand the population that has happened, I want you to change the debugging code a bit.

```
1 raise.contract.users.inspect
```

The last statement outputs the nested user forms that were created to handle the form submission. Admittedly, the `inspect` string is not beautiful but it quickly shows us some interesting facts.

```

1 [#<@fields = {"email"=>"wrong@format"}, 
2  @model = #<User id: nil, email: nil>, 
3  @_changes = {"email"=>true}]

```

This is an array of nested user forms. To keep it simple I only filled in one email address in the web form. As you can see, the `email` property is set and keeps the string I entered (line 1). Apparently, the form also tracks what fields were changed, but more about those implementation details at a later point (line 3).

What's more important is that the model we created in the `:prepopulate_if_empty` option is effectively wrapped by this nested form (line 2). The form keeps the model in the `@model` instance variable. And, as I promised you, no attributes have been set on the model, yet.

The form object indirectlys the input from the actual persisting model.

Another crucial thing to see is that the `User` model has been instantiated and wrapped inside the user form, as we just found out, however, the model has not been “physically” associated to the `Thing` model, yet.

```
1 raise contract.model.users
2 #=> []
```

Here, I access the contract’s model (the actual thing) and print the associated users, which turns out to be an empty array. That in turns means no users have been linked to the thing *model*, yet.

Now, what if I submit a form with an existing email address? Then the populator is gonna find the user and wrap it in a nested form. Check out what the validated form looks like.

```
1 raise.contract.users.inspect #=>
2
3 [#<@fields = {"email"=>"nick@trb.org"}, 
4  @model = #<User id: 1, email: "nick@trb.org">, 
5  @_changes = {"email"=>false}]
```

Indeed, the populator didn’t create a user but wrapped the existing model in the nested form (line 4).

This is exactly what Reform is designed to do. Create an object graph from arbitrary input, validate and process it and then, when you’re happy with it, persist it to the database by syncing the properties to models.

Saving

Speaking of syncing: we already have the remaining code in place to process the form and save it. This is the original code from chapter 5.

Let’s be crazy and do some more debugging. I want to see what the form does after I call `save`.

```

1 def process(params)
2   validate(params[:thing]) do |f|
3     f.save
4
5     raise f.model.users.inspect
6   end
7 end

```

I moved the “debugger” into the `validate` block and after the form’s `save` invocation (line 5). Doing so will raise the exception after all work has been done, and I’m expecting the same form, but this time the models should be persisted.

Note that I’m directly accessing the `Thing` model’s `users` and inspect those.

```
1 [<User id: 27, email: "paulo@trb.org"]
```

As expected, an array with one persisted user model. After calling `Form#save`, the form takes care of associating nested models to the parents, syncing and saving³⁵. I find this quite convenient, and also clean.

I hope the debugging didn’t scare you. It’s meant to help you understand how Reform and Trailblazer works, and that there’s no magic at all. At every step you can grab the respective object you don’t understand and introspect it.

This is fundamentally different to The Rails Way, where, once you invoke a public method, you completely lose track of what does what, as Rails doesn’t use objects with limited scopes but usually processes everything in one big class - the model.

Rendering a new form, validating and saving the nested object structure is now properly implemented. We have to use two options `:prepopulator` and `:populate_if_empty` but it makes sense to have two separate implementations here.

Adding Authorships

While we sit there and celebrate our success story of implementing a quite complex form, it knocks at the door. It’s one of our fancy business people. Oh no. Work.

³⁵Actually, a lot of the “model” logic like syncing and saving is implemented in the underlying twin of the form, and not in the form itself. The form acts as a thin orchestrating layer, only. From the outside, a form might appear as a monolithic monster that has “too many responsibilities”. We will discover twins in a later chapter, and learn how forms, representers, and twins work together.

And this time, it's "*really urgent*" requirement, a decision from the very top level, which is actually a "fix", because you, the developers, "*forgot to implement that*", maybe because it has never been specified. Whatever.

We need to add the concept of authorships. When adding a user to a thing, this should be marked as an unconfirmed authorship. Users have to manually confirm that they are authors of particular projects. And to make it extra hot: Users can only be added to maximum five things without confirming their authorship.

Luckily, we don't have a user section in our app, yet, which means we don't have a place where users can confirm their authorian existence. We need to implement the new association, though, and add some more validations. Five minutes, ok?

The authorship is supposed to associate a user and a thing, along with meta information like the confirmation date, status, and so on. This cries for a new model.

```
1 class Authorship < ActiveRecord::Base
2   belongs_to :thing
3   belongs_to :user
4 end
```

For a better understanding, here's an excerpt from the migration for this new table.

```
1 create_table :authorships do |t|
2   t.integer :user_id
3   t.integer :thing_id
4   t.integer :confirmed
```

As you can see, I added a `confirmed` field which represents the status of the authorship per user and thing.

I just learned about the `:through` relationship in ActiveRecord - and add it to Thing.

```
1 class Thing < ActiveRecord::Base
2   has_many :users, through: :authorships
3   has_many :authorships
```

The old (and apparently uncouth) `has_and_belongs_to_many` relationship is replaced with `has_many :through` association, both in the thing and user model classes. This allows to have a many-to-many relationship with the `Authorship` model in the middle.

I daresay that I like the way ActiveRecord does this and I was positively surprised that the old code kept working after I changed to this new association scheme.

Non-CRUD Behavior

So far, we don't need to change any code. When calling `save`, the form will still use the public writer `Thing#users=` when assigning the users collection to the thing. ActiveRecord will implicitly create authorships for us, or skip creating them, in case they do exist already.

We still need to set the created authorships to *unconfirmed*. While we could use a database default - and this is what you were thinking of, right? - let's do it by hand, just for the fun of it.

This manual step doesn't really fall into the typical CRUD category. Non-CRUD logic usually goes into the operation's `process` method.

```

1 class Thing::Create < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:thing]) do |f|
5       f.save
6       reset_authorships!
7     end
8   end
9
10 private
11   def reset_authorships!
12     model.authorships.each { |au| au.update_attribute(:confirmed, 0) }
13   end

```

Instead of using weird ActiveRecord callbacks or observers or whatever "best practice" the Rails Way's got in stock, I add a method invocation `reset_authorships!` into my `create`'s `process` method (line 6). Note that this additional logic is called *after* we let the form sync and save submitted data.

The implementation of this step goes into the same operation. At any point in an operation, you can retrieve the model using the `model` reader. I iterate all authorships of the thing we just created and use `update_attribute` to set the confirmed status to zero (line 12).

I don't get tired of repeating this, over and over again: Trailblazer does not have a problem with ActiveRecord or any other ORM per se. I just used its API to change persistent application state, and that is what I love ActiveRecord for. It is completely legit to use `update_attributes`, muck around with table fields and so on. As long as this code does not sit in the model itself, this is perfectly fine.

Complex Validations

We now have to make sure a user can only have a maximum of five unconfirmed authorships. Let's do this in a validation, cause that's what they were made for. As this validation is *per user* this goes directly into the nested user form.

```
1 collection :users,
2 # ...
3 validate :authorship_limit_reached?
4
5 def authorship_limit_reached?
6   return if model.authorships.find_all { |au| au.confirmed == 0 }.size < 5
7   errors.add("user", "This user has too many unconfirmed authorships.")
8 end
9 end
```

The `validate` class method allows us to add arbitrary validation methods to forms (line 3). Those methods will be invoked when the form gets validated, along with all the other formal validations you might have specified.

In the nested form, I add the `authorship_limit_reached?` method to implement the validation (line 5-8). Please note that all this code sits *in* the nested form's block. A nested form created with `collection :users do ... end` will simply create a new class for that form. We can add validations and methods without polluting the surrounding form or further nested ones.

I count if the authorships count is less than five and return if that's the case (line 6). No further action has to be taken if the count is less. Otherwise, I add an error message and mark the form as invalid (line 7). This will keep the operation from further processing.

Since this is all in a nested form class, `model` here refers to a user, not a thing! Remember, every form wraps around a model. We were just working in the nested user form, so the model is obviously a reference to the user.

Another validation that is still missing is limiting the maximum users per thing. This is not really business critical but I'll add it anyway, so the managers can (or can't?) blame me later.

```
1 collection :users,
2 # ...
3 end
4
5 validates :users, length: {maximum: 3}
```

To assert we're not exceeding the number of added authors and kill our poor database, I use another formal validation that caps the maximum user count to three (line 5). This validation sits in the main form as it counts the total amount of users, and is not applicable per user.

The used validator `length` is usually for strings but works great for collections, too. All it does is calling `users.size` and as `users` happens to be an array subclass it responds to `size`. A deep and profound way to validate collection size.

Isn't it a satisfying feeling, to have all validations in place? Will make us sleep well. But, hey, how do we know all those validations really work?

Right! Before we go on to the updating form, we need to test what we just implemented.

Testing Create

Tests still go into the `test/concepts/thing/operation_test.rb` file. We might think about splitting up once the file reaches the critical limit of about 200 lines of test code.

As done before, I won't discuss every test case in detail and skip a few. All tests are [in the repository](#)³⁶ and well commented.

```

1 class ThingOperationTest < MiniTest::Spec
2   describe "Create" do
3     it "invalid email" do
4       res, op = Thing::Create.run(thing: {
5         name: "Rails",
6         users: [{"email"=>"invalid format"}, {"email"=>"bla"}]
7       })
8
9       res.must_equal false
10      op.errors.to_s.must_equal "{:users.email=>[\"is invalid\"]}"
11      op.model.persisted?.must_equal false
12
13    end

```

The first test asserts the email validations. I run `Thing::Create` with two invalid email addresses (line 4-8) and then make sure validations complain about the wrong emails (line 11). To make sure nothing has been written to the database, I also inspect the operation's model (line 12).

In the next test case the `:skip_if` option is tested. You might be wondering whether this is a redundant test, and the answer is actually “yes”. This option is well tested in Reform itself, nevertheless, let's write a quick test for educational purposes.

```

1 it "all emails blank" do
2   res, op = Thing::Create.run(thing: {name: "Rails", users: [{"email"=>""}]})
3
4   res.must_equal true
5   op.model.users.must_equal []
6 end

```

Running the operation will work and create a new thing instance (line 2). By testing that `model.users` returns an empty list I assure no stale ghost users have been created.

The following test creates a valid thing along with one new and one existing user.

³⁶https://github.com/apotonick/gemgem-trrb/blob/chapter-07/test/concepts/thing/operation_test.rb

```

1 it "valid, new and existing email" do
2   solnic = User.create(email: "solnic@trb.org") # TODO: replace with operation.
3
4   model = Thing::Create.(thing: {
5     name: "Rails",
6     users: [{"email":>"solnic@trb.org"}, {"email":>"nick@trb.org"}]
7   }).model
8
9   model.users.size.must_equal 2
10  model.users[0].attributes.
11    must_equal("id":>solnic.id, "email":>"solnic@trb.org")
12  model.users[1].email.must_equal "nick@trb.org" # new user created.
13
14  model.authorships.pluck(:confirmed).must_equal [0, 0]
15 end

```

For setup, I create an existing user (line 2). Note the TODO at the end of the line. I use ActiveRecord directly, as we do not have an operation, yet, to create a user.

I then run the operation, as already mentioned, with the existing user's email and a new one (line 4-7). The newly created model must have two users now (line 9). It's not that important to test that the first user, the existing one, has the same attributes as it had before. This is more me being paranoid that things might have gone wrong (line 10-11).

The second user should have the email address we provided (line 12). And as a last constraint, both created authorships must have a zeroized confirmed field (line 14). You haven't forgotten that both user and thing point to the authorships, so I can safely use thing.authorships here to retrieve the information I need.

Another test will check that we can only create three authors as a maximum.

```

1 it "users > 3" do
2   emails = 4.times.collect { |i| {"email":>"#{i}@trb.org"} }
3   res, op = Thing::Create.run(thing: {name: "Rails", users: emails})
4
5   res.must_equal false
6   op.errors.to_s.
7     must_equal "{:users=>[\\"is too long (maximum is 3 characters)\\"]}"
8 end

```

The times method strikes again and helps to setup four emails (line 2) which I pass into the operation call (line 3). When checking the error message you can see that it says "characters" instead of "users" (line 6-7). This is due to the length validators nature of testing strings, normally. As this validation

breach will never happen under normal conditions, I don't care. The likeliness that someone hacks my HTML form to add another user shall be rewarded with this funky error message.

In the last test we examine one of the last validations we added: A user can only have a maximum of five unconfirmed authorships.

```

1 it do
2   user = {"email"=>"nick@trb.org"}
3   5.times { |i| Thing::Create.(thing: {name: "Rails #{i}", users: [user]}) }
4   res, op = Thing::Create.run(thing: {name: "Rails", users: [user]})
5
6   res.must_equal false
7   op.errors.to_s.must_equal(
8     ":\"users.user\"=>[\"This user has too many unconfirmed authorships.\"]}")
9 end

```

Can you tell that I get into using `times` to generate test setups? I run the create operation five times, with a different name and the same user each time (line 3). This will create five things with the same user. After running, the user will have five unconfirmed authorships. The sixth time, still with the same user, should fail as this doesn't comply with the management's desires (line 4-8).

I love how factories completely become redundant. Since I use Trailblazer, I don't even think of test factories anymore, the environment setup strictly use the application's public API. And those are operations.

Presentation Tests

To complete this section, we need to add two more tests to the thing integration test at `test/integration/thing_test.rb`. Proper rendering of the extended form has to be verified.

First, the form when rendered in `/things/new` context is examined.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "allows anonymous" do
3     visit "/things/new"
4     # ...
5     page.must_have_css("input.email", count: 3)

```

When visiting the form to create new things, we should see three input fields in order to add users (line 5). The same should happen when viewing a submitted, invalid form. I added a test but save the paper here, since it is identical.

Updating Things

We should also allow users to change their author selection after creation. Quite productive, as we are, we already implemented the “edit thing” operation in chapter 3, and you surely remember that the only change from the create form was to disable users from changing the thing title. We implemented that by making the property readonly.

While I do want to provide additional email fields to add authors in hindsight, existing authors should be readonly. Instead of allowing to change their email address, which doesn’t really make sense, I’d rather add a button to remove the user from the thing.

This won’t delete the user from Gemgem, but only remove the particular authorship.

When adding new authors through the form we’re working on, they will be persisted as users in the database. In the next chapter, we will send out an email to them, asking them to join Gemgem. They might or might not see a list of authorships to confirm, then. As I just said, clicking the “Remove” checkbox in our edit form won’t touch the user, it will delete the authorship, only.

Hacking the View

In order to provide a “Remove” button in the edit form, we need to change the form’s view in `app/views/things/new.html.haml`.

Remember, the form view is used for both creating and editing. The sad news here is, we will use several helper methods in the view from the update form. Those methods are only available in the update form so far. This is why I have to introduce this incredible ugly switch.

```
1 %legend Do you know any authors?
2 = f.fields_for :users do |a|
3   -if @operation.instance_of? Thing::Create
4     = a.input :email
5   -else
6     = a.input :email, readonly: a.object.removeable?
7
8   -if a.object.removeable?
9     = a.input :remove, as: :boolean, input_html: {checked: false}
```

As you may have noticed, I’m using methods like `removeable?` directly on the form object. Since I will add this `removeable?` method only to the *update* form, this view - without the switch - would break when presenting the *create* form.

And, yes, this is horrible code. I mean, I don’t have problems with deciders in views, but this just makes my eyes bleed. This is to make you feel the pain of Rails’ lack of a proper view layer

with polymorphic concepts. Later, not in this chapter, though, this form will become a true view component using view inheritance to tackle with different contexts.

For now, bear with me, and accept that I just used `if/else` and `instance_of?` in a view. Let's walk through it step by step.

First, I use the `@operation` instance variable to decide the context of this view. By using `instance_of`, and I hate it, I find out whether to render the create or the update fields (line 3).

Normally, in create mode, we will simply render an email field (line 4).

When updating, things get a bit more tricky. I still add the email field, but make it visually readonly when `a.object.removeable?` returns true (line 6). This mysterious invocation happens again, and when true, adds another field to the form named `remove`. The `remove` field is a checkbox, and unchecked per default (line 8-9).

What is going on here, and especially what is this `removeable?` call about?

Helpers in Forms

I am using `fields_for` to render the three email fields. This form builder method will iterate over the thing form's `users` and yield the `a` object to the nested block. This `a` is another form builder object that wraps a user form. By invoking `a.object` I get access to the nested user Reform object.

Again, this is really clumsy and not ideal, in my opinion, but for now it has to do the job. Just keep in mind that `a.object` returns a nested user form object, and every method we invoke will be called on the user form.

In this example we use the `removeable?` method. This method doesn't exist in the form and we need to implement it. We need to jump back into the `Thing::Update` operation's contract.

```

1 class Update < Create
2   # ...
3   contract do
4     property :name, writeable: false
5
6     collection :users, inherit: true do
7       def removable?
8         model.persisted?
9       end
10    end
11  end
12 end

```

The `removeable?` method is implemented in the `users` property (line 9-11). This will allow calling this method for every nested user. So, when calling `a.object.removeable?` this very method will be invoked!

In this method I simply check whether the wrapped user model is persisted (line 10). I can access the form's model at any point using `model`, and I use ActiveRecord's `persisted?` on it to find out whether this nested form is representing an empty user or showing the actual email of an existing one.

Only existing users, or authorships, should be removable, and in this case, `persisted?` does the trick.

Let's recap. The update form shows a bunch of filled out email fields, each representing an existing user that was added as an author earlier. The update form might also render a few empty email fields, depending on how many authors are already added.

The “Remove” checkbox shall only appear for “real” authors, that are already added to the thing. A decider `removeable?` is introduced to find out whether or not this form is wrapping a “real” user.

This decider is then used in the view to render, or not render, the checkbox to remove. It's too easy.

Form Inheritance

I am pretty sure you've been wondering the last few minutes about the `:inherit` option, and where all the properties from the nested `users` form have gone to.

```
1 collection :users, inherit: true do
2   # ...
3 end
```

We're now getting straight down to the nitty gritty of forms and inheritance.

The `Update`'s operation form is inherited from the `Create` operation form. Trailblazer does that internally when you subclass an operation. Inheriting a contract means all the properties and nested forms, the helper methods you added, the validations, and so on will be inherited, or copied, to the new class.

In an object-oriented environment it is desired to override specific parts of the inherited class. And that's what I do when defining the new `collection :users` form. I override this part of the original form.

However, when I omit the `inherit: true`, the entire nested form class will be overridden with the new class I just specified. This means that we will not only lose all the properties defined in the original class, but also validations.

You can play around with this option to understand it better now. Remove the `:inherit` option and see what happens.

```
1 collection :users do
2 end
```

The rendering of the update form will crash now, as the `email` property couldn't be found. Now, add the property manually.

```
1 collection :users do
2   property :email
3 end
```

Now the form will render, but you'll see that you lost all the validations, and so on.

When confused, always try to picture the `:inherit` option like the `super` call in Ruby, because that inspired me when I designed it.

When you override a method, it's gone. Your new method implements this method now. This is when you call `property` without the `inherit` option in a derived contract.

However, once you use `inherit: true` it's a bit like the `super` method, in a declarative meaning. It says "*Let me override this directive, but inherit all options, nested properties, and so on, from the original property!*".

The `:inherit` option is a powerful concept to allow real object-orientation on a declarative level. A lot of work has gone into designing, implementing and testing this feature and many projects benefit from it with clean, DRY forms for different contexts.

Virtual Properties

Having the inheritance part in place, we should try and render the update form. It will raise an exception, as the form builder requires the `remove` property. This error comes from the view.

```
1 = a.input :remove, as: :boolean, input_html: { checked: false }
```

When rendering the form, the form builder queries for every field it is assigned to render. This means we can either add an empty `remove` method in the nested user form, or use a virtual property, which does exactly the same.

```

1 collection :users, inherit: true do
2   property :remove, virtual: true
3
4   def removable?
5     model.persisted?
6   end
7 end

```

Now the user form got two properties. The `email` is inherited, the `remove` property is added and casted into a virtual field. Virtual fields are not readable and not writeable, meaning Reform neither will ask the model for the value of this field, nor will it try to write the new value back to the model when syncing.

Often, virtual fields are used for password confirmation fields and the like.

The form now renders, and, hey, cool, for existing authors it shows a remove checkbox. And now the sensational feature: clicking a checkbox and submitting the form should unlink the user from the thing and delete the authorship, while letting the user alive!

Populator

"I clicked the remove checkbox for an author when editing a thing, punched submit and it didn't get removed, it's still there." This, or a similar message, will be the first bug report in your inbox should you have shipped Gemgem right now.

Of course, the remove checkbox is something we invented just now - this is not a feature of Reform. We have to write that functionality ourselves. Nevertheless, Reform makes it pretty simple to implement it, so let's make the remove button actually do what it says.

Removing users only works from an editing form, which is processed by `Thing::Update`. Implementation happens in its contract in `app/concepts/thing/contract.rb`.

```

1 module Thing::Contract
2   class Update < Create
3     # ...
4     collection :users, inherit: true, populator: :user! do
5       # ...
6     end
7
8   private
9   def user!(fragment:, index:, **)
10    if fragment["remove"] == "1" # don't process if it's getting removed!
11      serialized_user = users.find { |u| u.id.to_s == fragment["id"] }
12      users.delete(serialized_user)

```

```

13      return skip!
14    end
15
16    return skip! if users[index] # skip if already existing
17    users.insert(index, User.new)
18  end
19 end

```

You're about to learn about the `:populator` option. This is the most powerful and flexible directive to fine-tune your deserialization. In fact, `:populate_if_empty` is a macro implemented using this option, and hence provides a subset of its features, only.

```

1 module Thing::Contract
2   class Update < Create
3     #
4     collection :users, inherit: true, populator: :user! do

```

By providing `populator: :user!` I configure a populator method that will be called for each `users` fragment in the incoming hash (line 4). While I could also write my populator as a proc, I prefer methods.

Now, don't confuse that with the `:populate_if_empty` option. The latter one is only called when Reform can't match the incoming fragment to a nested form. The `:populator` is *always* called and you have to make sure the correct nested form is returned. While this is more work, you have control over the entire deserialization process.

The method implementing the populator needs to be an instance method of the form defining the property.

```

1 collection :users, populator: :user! #..
2
3 def user!(fragment:, index:, **)
4   if fragment["remove"] == "1"
5     serialized_user = users.find { |u| u.id.to_s == fragment["id"] }
6     users.delete(serialized_user)
7     return skip!
8   end

```

As all dynamic options in Reform, it receives an options hash as the only argument. I use keyword arguments to extract the `:fragment` and the `:index` key (line 3).

Populator: The fragment

The `fragment` hash is of peculiar interest here. For a newly submitted author, this will look as follows.

```

1 def user!(fragment:, **)
2   fragment #=> {"email"=>"kasia@trb.org"}

```

This is the extracted fragment from `params` that matches to this form.

For an existing author with an authorship, this hash might look a little bit different.

```

1 def user!(fragment:, **)
2   fragment #=> {"id"=>"1", "email"=>"nick@trb.to", "remove"=>0}

```

The fragment now contains the user's id and the remove flag, too. While the remove field comes from the actual form input field we added earlier, the id is added automatically by Rails' form builder.

With this in mind, recall the upper part of the `user!` method. If the `remove` field is set, we trigger the delete behavior (line 4). I grab the fragment's `"id"` value and find the respective user (line 5). Note that I can use `users` here, which resolves to `form.users` as this is an instance method of the form. That way, I simply iterate the form's users and find the delete candidate - in my last input example, this would be the user with email `"nick@trb.to"`.

I then call `users.delete` and pass in the candidate (line 6). This will "physically" remove that item from the form's users. However, and now stay calm, this doesn't change any persistent data, yet. This mysterious `"nick@trb.to"` user is only deleted from the runtime object graph. The form will remember that, though, and apply persistent changes when you call `save`.

Populator: `skip!`

After the user has been removed, I return `skip!` (line 7). This will stop execution of the populator method. At the same time, it will tell Reform to stop deserializing this fragment, and move on to the next fragment, just to start the same cycle of deserialization again.

The `skip!` instruction, and note that you have to return it, is extremely helpful for customized deserialization. You can use it in `:populator` to stop further processing of a fragment. For example, say you had a validation that should happen upfront, before the object graph is being generated, and that should suppress certain fragments, `skip!` is your friend.

Now, the remaining part of the populator.

```

1 def user!(fragment:, index:, **)
2   # ...
3   return skip! if users[index]
4   users.insert(index, User.new)
5 end

```

When deserializing a collection fragment, Reform knows where it is. As the populator is called for every item fragment, the populator receives the index. After we've ruled out this item is not a deletion, I check if there is already a user existing for that fragment (line 3). If that is the case, I also skip the rest of its deserialization.

I do this because we do not allow updating any of the user's attributes via this form. By skipping if user exists, the fragment is discarded and everything on the nested form will remain the same.

If the fragment does not represent a deletion, and no user corresponds to it, I add a new User instance at position `index` to the form's `users` collection (line 4). The return value of this statement is important, that's why it is the last line of the method.

You might be wondering how we handle the `:all_blank` function? The `:skip_if` option is inherited from the original contract. And, since this option is evaluated before the populator, empty fragments are excluded from deserialization automatically, we don't have to consider this case in the populator.

Persisting the Deletion

Before we discuss how this deletion gets propagated to the database, I invite you to some more debugging. This is how I learned programming.

Create a thing with two authors, and then edit it. Now, tick one of the two authors for removal. In `Update#process` add a raise *before* you call validate.

```

1 class Create < Trailblazer::Operation
2   # ...
3   def process(params)
4     raise contract.users.inspect
5     validate(params[:thing]) do |f|
6       f.save
7     end
8   end

```

In the exception you will identify two nested user forms. This makes sense as no deserialization has happened, yet, and we see the original edit form.

```

1 [
2   #<@fields={"email"=>"ryan@trb.de"}, @model=#<User id: 1, email: "ryan@trb.de">,
3   #<@fields={"email"=>"nick@trb.de"}, @model=#<User id: 2, email: "nick@trb.de">,
4 ]

```

When moving the raise into the validate block, and supposed you checked the second author for removal, here's what the form graph now looks like.

```
1 [  
2 #<@fields={"email"=>"ryan@trb.org"}, @model=#<User id: 1, email: "ryan@trb.org">  
3 ]
```

The `users.delete` from the `user!` populator literally *deleted* the nested form from the object graph. How does this get persisted, though?

It's simple. When calling `save` on the form, it will sync all its properties back to the original `Thing` model. This implies calls like `thing.title="Rails"`, or `thing.description=""`, but this also assigns nested properties.

Given the example scenario I just used, the `users` property invocation will look as follows.

```
1 thing.users = [#<User id: 1, email: "ryan@trb.org">]
```

As can be seen from this snippet, Reform (actually, this happens in `Disposable`) simply pushes the nested form's models to the collection writer. And since we *deleted* the second nested form, the collection synced back to the model does only consist of the first author.

The great thing about the syncing is: `ActiveRecord` will take care of the rest. It will delete the authorship for us automatically when updating the `users` property with the new, smaller collection.

Both the create form and the update function work exactly the way we want now. Users can be created, validations make sure they don't exceed certain limits. Empty forms are ignored. Existing fields can't be changed.

Let's write a few quick tests for the `Update` operation.

Testing Update

Following the same path that we did for the create tests, I start with a rendering test of the edit form in `test/integration/thing_test.rb`.

```

1 class ThingIntegrationTest < Trailblazer::Test::Integration
2   it "edit form shows readonly author" do
3     thing = Thing::Create.(thing:
4       {"name" => "Rails", "users" => [{"email" => "joe@trb.org"}]}).model
5
6   visit "/things/#{thing.id}/edit"
7   page.must_have_css "form #thing_name.readonly[value='Rails']"
8   page.must_have_css(
9     "#thing_users_attributes_0_email.readonly[value='joe@trb.org']")
10  page.must_have_css "#thing_users_attributes_0_remove"
11  page.must_have_css "#thing_users_attributes_1_email"
12  page.wont_have_css "#thing_users_attributes_1_remove"
13 end

```

The fixture consists of a new thing with one user (line 3-4). I then request the edit form for the thing we just created (line 6). The thing name has to be readonly (line 7).

On the page, the first email field has to contain the correct email address from the existing user, and it must have the class `.readonly` (line 8-9). Also, the remove button has to be present (line 10). I then check for another email field just to make sure pre-population worked (line 11). This should not have a remove button, yet, as this represents a blank email field (line 12).

After the rendering is sorted we quickly focus on extending the `Thing::Update` tests in `test/concepts/thing/operation_test.rb`. I won't go into every test case here as it becomes a very tedious and space-consuming task to discuss test cases that are sitting nicely documented on Github.

My favorite test case is the one where we make sure existing emails can't be changed.

```

1 describe "Update" do
2   #
3   it "doesn't allow changing existing email" do
4     op = Thing::Create.(
5       thing: {name: "Rails ", users: [{"email"=>"nick@trb.org"}]})
6
7     op = Thing::Update.(id: op.model.id,
8       thing: {users: [{"email"=>"wrong@nerd.com"}]})
9     op.model.users[0].email.must_equal "nick@trb.org"
10 end

```

First, I create an existing user (line 4-5) using our fantastic operation. I then use the `Update` operation where I inject a different email address (line 7-8). After the operation has been run, the email has to be identical to the old, original email address (line 9).

This test demonstrates how incredible simple it is to write edge-case tests with operations and forms. A nightmare, when I think back how that had to happen in traditional Rails with a mix of fixtures,

controller- and model tests that probably had to change global state variables to achieve a *almost-production* environment.

The other test case I want to discuss is when removing existing authors.

```

1 it "allows removing" do
2   op = Thing::Create.(  

3     thing: {name: "Rails ", users: [{"email":>"joe@trb.org"}]})  

4   joe = op.model.users[0]  

5  

6   res, op = Thing::Update.run(id: op.model.id,  

7     thing: {name: "Rails", users: [{"id":>joe.id.to_s, "remove":>"1"}]})  

8  

9   res.must_equal true
10  op.model.users.must_equal []
11  joe.persisted?.must_equal true
12 end

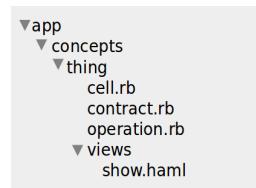
```

All I have to make sure is that I pass in the correct input into the update operation (line 6-7), especially the "remove" key makes sense here. After the operation, the thing's users collection must be an empty list (line 10).

An important check after that is to make sure the user object still exists and wasn't erased from the database. I use the persisted? method for that on the original author that I saved earlier (line 4 and 11).

Extracting Forms to Separate Files

That chapter has been great so far. We learned a lot about forms, nesting, object graphs and all the cool possibilities we have with form objects.



The form object is a pivotal element in every application. Deserialization, setting up an object graph and validating it is the heart of the kind of software we write. This might result in form classes becoming bigger and more complex. At some point, and embedded in the operation class, this might be hard to work with.

Luckily, we can simply use a separate file for our forms, since the contracts in operations are just plain classes.

I add a new file in `app/concepts/thing/contract.rb` and basically copy and paste the contracts into it.

```

1 module Thing::Contract
2   class Create < Reform::Form
3     model :thing
4
5     property :name
6     #..
7   end
8 end

```

As the first thing, I define the module `Thing::Contract` (line 1). I then introduce a new class `Create` which is derived from `Reform::Form` (line 2).

Basically, the structuring for external Trailblazer objects should be `Concept::Technology::Environment`. We will later do the same structuring with representers, in chapter 11 and 12, where we have `Thing::Representer::Create`, and so on.

In the new `Create` class I literally just cut and paste the contract `do ... end` block from the original operation, with one exception: you have to call `:model` manually to name your class for the form builder (line 3). Another convenient thing that happens automatically in an operation.

In order to use this form, a minimal change in the operation is required.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     contract Contract::Create

```

Instead of defining the form here, I assign the form to the operation via the contract DSL method (line 3). Note that you could reference any form you want here with namespaces and what not.

For consistency, let's do the same for the `Thing::Update` operation.

```

1 module Thing::Contract
2   class Create < Reform::Form
3     #..
4   class Update < Create
5     property :name, writeable: false

```

Normally, the operation will inherit the contract automatically for us from its parent class. Here, we have to do this manually for `Contract::Update` (line 4). As we use inheritance, we do not have to call `model` again in the contract, this is Reform's deriving behavior. We can simply move the contract code to the new class (line 5).

```
1 class Thing < ActiveRecord::Base
2   class Update < Create
3     contract Contract::Update
```

I do not have a strict rule, yet, when to apply the form extraction to operations, but as soon as the contract has more than a few lines, I usually introduce a separate file.

Wheew! We're Masters of the Form now!!! Let this sink for a bit, and have a cold beer. In the next chapter we're going to learn about callbacks and operations, and how to deal with more than one model per operation. Also, and this is gonna be quite cool, we'll dive into view caching and how operations can expire caches.

Callbacks

In the last chapters we have focused on object validation and creation which was mostly encapsulated by the form twin. In real applications, you need more than that, though. After the initial object graph has been created, post-processing takes place. Things like file uploads or sending out notifications, known as *callbacks*, have been established as a concept of web development.

Callbacks in Trailblazer go further than that and might surprise you in the way they differ to the conventional Rails Way.

Code for this chapter you can find [in the repository branch³⁷](#).

Rails Callbacks

The idea of callbacks in ActiveRecord and controllers is helpful. It allows to execute a number of callbacks at well-defined points in the lifecycle of an object. In addition to that, callbacks can be added in a declarative way without having to override methods. You simply add extending behavior by calling `after_validate`, and so on.

With the ease comes trouble, of course.

First of all, the callback hooks Rails provides are very high-level. That alone is a good thing. It allows to extend the flow for the entire object. It does not provide a way to hook into separate properties, though. You can't attach a callback that only gets executed for the `email` field of a user, for instance.

That is because Rails doesn't buy into the idea of schemata. A callback doesn't even know a field `email` exists as everything is inferred from the database table at runtime. Every single callback has to write their own filter logic to limit its scope to a property. Countless times I've seen logic as follows.

```
1 def notify_author!
2   return if email.nil?
3   send_email!
4 end
```

This creates an immense load of redundant logic, noise, and makes it hard to find out what is really the essence of a callback. Somewhere between the filter logic sits the actual code you're interested in.

The other big problem with callbacks is the lack of execution context. Callbacks getting in your way by being triggered when you don't want them. Callbacks always get invoked, once defined they will be run everywhere and under any circumstances unless you take action - hacks - to prevent that.

³⁷<https://github.com/apotonick/gemgem-trrb/tree/chapter-08>

Where different contexts in Trailblazer usually mean subclasses or declarative overriding, in Rails this is half-heartedly implemented with conditionals. All happening in the same class, people use `:ifs` to prevent callbacks from being run at certain occasions. Of course this never works as it should and you end up in “*Rails Callback Hell*”, evidenced by literally hundreds of blog posts.

Controller Filters

A common problem I have when jumping onto refactoring an existing Rails application is figuring out what callbacks are invoked in which request. The issue is not only understanding the model callbacks, but also what filters in a controller do.

Controller filters in Rails can be categorized into `before_filter`, which are run before the actual code in the action method, and `after_filter`. I have never seen anyone using an `around_filter` and I don’t even know how they work.

Filters were originally meant to *filter* request integrity, hence the name `filter.before_filters` were added to intercept unauthorized requests, `after_filter` was then provided to clean up the mess you made in your controller action.

The name itself manifests the problem about filters: they shouldn’t be used for anything but authentication and maybe authorization. Of course, this isn’t stated anywhere in the Rails books, and people use filters to do just anything in their controllers: authentication, authorization, munching the `params` hash, validations, invoking callbacks, persistence... I’ve seen any kind of logic in `before_filters` you can think of.

Most filters are business logic and have no place in the controller. In Trailblazer, most of the filter logic is moved into the operation scope. We use `before_filters` only, and limit their usage to authentication. This is the reason I won’t discuss filters in this chapter but the following, where we learn how Trailblazer does authentication.

Domain Callbacks

We spent quite a while on creating a user interface to allow users setting up a *thing*, add authors, and comment on things. As a next requirement, we need to notify users about their new authorship. When adding a user to a thing this should be communicated in a nice email being sent out to the author telling them “*Hey, BTW, you were added to that thing!*”.

This is too easy, and you as the developer shouldn’t say that aloud, because now the business adds another requirement. Depending on whether the user was just created or not the email has to contain an additional welcoming section informing the unsuspecting email owner that they were not only signed up for Gemgem and are now a valid user, no, they are also a thing author.

Remember that non-existent users are created ad-hoc when they’re added to a thing? Yepp, that means the email has to contain welcome and authorship text.

Not that I want to go into *The Rails Way* ever again, but to give you a better understanding about why I chose to replace callbacks in Trailblazer, let's quickly discuss how we'd do the above requirement in Rails.

Callbacks in Rails 5

When it comes to emailing freshly created users, every Rails developer will instantly think of an `after_create` callback in the `User` model.

```
1 class User < ActiveRecord::Base
2   after_create :notify_author!
```

So far, this looks perfect. It is one line of code (without the actual logic) to be run after a new user got persisted. This could generate the welcome section for the email. Also, assuming that users are only created when being added to things, this callback could be used to generate the entire email.

Now, how do we track when an existing user gets added as an author? This needs to send out an email, too. We push that into the `Authorship` model, of course.

```
1 class Authorship < ActiveRecord::Base
2   after_create :notify_author!
```

In this callback, we encounter our first problem with Rails callbacks. This code is run whenever an authorship is created. We now have to make sure this logic gets run only with an existing user, since the new user is already handled in the `User` class.

Even though we only have two callbacks, it starts to get complicated. I don't want to say "messy", yet.

Immediately the next problem emerges. In the next chapter users will be able to sign up via a separate form. This will trigger the `User`'s `after_create` hook, too, and this is wrong, because manually signed up users need a completely different email.

Yes, we could now patch the existing code in `User` to differentiate between the two contexts, and so on, but I am not going to do this. The scenario I just illustrated is still manageable but I already hate it. Logic to handle callbacks is distributed to the persistence tier, spread over several classes, contains too much knowledge about what is going on where, and is incredibly hard to understand.

Rails callbacks are probably the worst concept in the framework. They really *make* developers do it wrong. Unfortunately, this hasn't been addressed in Rails core, yet, and I am not sure why, because even core members hate callbacks.

Ad-hoc Callbacks

I decided to completely replace callbacks with a mechanism separated from persistence and encapsulated in its own class. What works with view models, forms, representers, or operations will most probably also work with business dispatching logic.

In order to write clean callback logic, we have several alternatives in Trailblazer. I start illustrating the simple one and will then explain the high-level Trailblazer solutions, which are still simple, but use additional code to allow you a declarative setup of nested callbacks and an imperative execution mechanism.

Let's get back to the basics: Operations embrace an action of your application in a class. The most conclusive next step when adding callbacks to that action is: add the callback logic to your operation!

What sounds primitive and naive is actually the simple solution to the aforementioned problem. When adding callback logic to your operation, you make sure this code is run only in this context. An operation represents a context, so it must be the right place for dispatch logic. In addition to that, the callback code is defined in one single place. Let me show you what I mean.

```

1 class Thing::Create < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:thing]) do |f|
5       f.save
6       notify_authors!
7     end
8   end
```

My callback is a simple invocation of a new method `notify_authors!` in my operation (line 6). This method is only called when the form's valid and got saved. Note that I call it *after* the `f.save` line, making it an explicit `after_save` callback without having to clutter logic over models. This single line is the dispatch, the business logic will happen in that method.

```

1 class Thing::Create < Trailblazer::Operation
2   # ...
3   def notify_authors!
4     model.users.collect do |user|
5       NewUserMailer.welcome_email(user)
6     end
7   end
```

To give you an idea how this “ad-hoc” callback could look like, I implemented a sample `notify_authors!` method in the create operation. Since we have access to the model and the contract I

directly access the `model`'s users and iterate them (line 4-6). For each user, I send out an email using an imaginary mailer (line 5).

This is of course not doing exactly what we need to do. I hopefully managed to show you how simple it is to add callbacks to Trailblazer operations by simply invoking methods right where you want them to be invoked. Nevertheless, my callback doesn't send out different mails dependent on the user's creation status.

Unfortunately, there is no way in ActiveRecord to find out if a model was just created after saving it. We could simply collect this information before we save the form in the `process` method. This is unnecessary, though, since the form object provides some cool features you're gonna love in combination with Trailblazer's callbacks.

The Persisted Module

To send out the correct emails to our thing authors we need find out whether they were created in the process of the `Create` operation, or whether they were already existing.

Every form is a `Twin` object with validation semantics. In a form, most of the work is done by the `twin` - a concept we didn't officially get introduced to, yet. In fact, you've been working with twins throughout the book! Every time we were playing with forms, we were using twins, I just didn't tell you.

It is now time to learn about twins and how they help us in callbacks.

As you may have noticed, the API of a form is extremely simple and breaks down to the following three steps.

1. When instantiating, the form populates its properties with the values from the decorated model. This is actually behavior inherited from `Twin` and not implemented in Reform.
2. After instantiation of a form comes validation. The `validate` invocation will update property values and possibly add more objects to a nested form property. This is the only function provided by Reform.
3. When validation was successful, data gets written back to the models using the `save` method. Again, this is implemented in `Twin`.

This super simple life-cycle of a form, or a twin, makes it elementary to track changes. For example, we can easily find out if a model got persisted during the `save`. This works by taking advantage of the `Persisted` module which is included automatically in every form.

```

1 thing = Thing.new
2 form = Thing::Form.new(thing)
3
4 form.persisted? #=> false
5 form.save
6 form.persisted? #=> true

```

The first two lines are usually done implicitly in the operation (line 1-2). Every form, yes, even nested forms, respond to `persisted?` and will report the model's persistence status by simply invoking the model's `persisted?` reader³⁸. It is obvious that the first query will return `false` and after saving, the same method will report a persisted model (line 4 and 6).

Now, that alone won't help us as we still don't know if the user was created just now or months ago. The innovative trick is that twins track field changes. Using this, the `Persisted` module gives us another helpful query method.

```

1 form.save
2 form.created? #=> true

```

The `created?` method is great and has helped me a lot to write callbacks of all kinds. It simply checks if `persisted?` has changed when calling `save`. If yes, this implies the decorated model has just been created. If not, this means the model was already persisted and thus the `save` was an update, not a creation.

Another example showing the opposite behavior will make it more understandable.

```

1 thing = Thing.find(1)
2 form = Thing::Form.new(thing)
3
4 form.save
5 form.created? #=> false

```

Repeating the same workflow with an existing thing, the `created?` query will return `false`. This is because even after saving the form, the `persisted?` flag doesn't change as the model was already persisted.

It's important to include the `Persisted` feature into the contract, should you rely on `on_create` callbacks.

³⁸The `persisted?` method is a concept found in every standard ORM. This allows using the `Persisted` module in any environment, without being limiting to ActiveRecord.

```

1 module Thing::Contract
2   class Create < Reform::Form
3     feature Disposable::Twin::Persisted

```

Including `Persisted` will now assume your contract's model responds to `persisted?` (line 3).

Explicit Callbacks

Now, let's apply this new knowledge to the problem we were trying to solve. By using `created?`, we can implement our first real callback in the create operation.

```

1 def notify_authors!
2   contract.users.collect do |user|
3     if user.created?
4       NewUserMailer.welcome_email(user.model)
5     else
6       ExistingUserMailer.added_email(user.model)
7     end
8   end
9 end

```

A few things have changed to the `notify_authors!` method. Firstly, note that I no longer work on the model, but on the contract (line 2). Iterating the contract's users will provide me the form objects with the twin semantics I need.

Now, I can actually use `created?` to query if the user just got created or has been in the system before (line 3). Keep in mind that this is called after we called `save` on the form. The twin simply looks at the `persisted?` field and checks if it has changed since the form was instantiated, nothing more.

Having found out the essentials, I can dispatch to the mailers which handle the existing and new user case (line 4 and 6). We will quickly implement the mailers at a later point. Right now I want to focus on callbacks, dispatching and differentiation of application state.

The next task is to re-add our `reset_authorships!` function from the last chapter. We needed to run this method to set all created Authorships to unconfirmed.

```

1 def reset_authorships!
2   model.authorships.each { |aship| aship.update_attribute(:confirmed, 0) }
3 end

```

In `reset_authorships!`, we blindly go through all users attached to the thing, grab the authorship models associating the two, and mark them unconfirmed. Indeed, this is a bit violent, almost brutal, and this works for creating, but will get us in trouble when updating things.

Anyway, we'll fix this in a minute. Let me now show how I add the explicit callback in the operation.

```
1 def process(params)
2   validate(params[:thing]) do |f|
3     f.save
4     notify_authors!
5     reset_authorships!
6   end
7 end
```

You guessed right, this simply is another method invocation in the operation's `process` method. By running the callback after saving the form, I make sure the environment is as expected and all the `Authorship` instances are persisted (line 5).

Explicit Callbacks: Inheritance

So far, we added two callbacks to the `Create` operation. One to send out notifications to authors while respecting their membership status. Only freshly added authors get the extended version of the notification email.

The second callback resets all authorships to unconfirmed state.

In order to see the problem that arises, we need to go one step further. I want you to remember that we not only have a creating operation, but also an `Update` one that inherits from `Create`. This inherits the `process` method and will run the exact same set of callbacks for updating things.

For the notifications, this is still going to work. The callback we wrote earlier will send out notification emails to added users, just as we did in `Create`.

However, the second callback `reset_authorships!` is wrong. When run in `Update` context, it will reset all authorships, even the ones we created earlier. Imagine we had a way to confirm authorships, yet. The callback would ruthlessly reset every associated authorship to unconfirmed, as it doesn't check any state but runs through the collection like a berserk.

While this is not a real problem right now because we can't confirm authorships anyway, yet, we have to fix it.

I could overwrite `reset_authorships!` in the `Update` operation and leave the original implementation in `Create` as it is. This would show another advantage of the inheritance philosophy we use in Trailblazer.

The actual trick I'm gonna use to fix this problem will work in `Create`, too, so I will correct the behavior for both operations at once using another feature from twins.

```
1 class Thing::Create < Trailblazer::Operation
2 # ...
3 def reset_authorships!
4   contract.users.each do |user|
5     next unless contract.users.added.include?(user)
6     user.model.authorship.update_attribute(:confirmed, 0)
7   end
8 end
```

Before running through this snippet, let's reflect what we need to do. Every time an authorship is created, which means a user is associated to a thing, we need to reset its `confirmed` field to zero. This should only happen once in the authorship's life! In no way should we reset it again, especially not when the `confirmed` flag has flipped.

As promised, another trick from the Twin API is exposed here.

In line with the first callback example, I work on `contract` in the callback method, as this gives me the twin API (line 4). I loop over the `users` for this particular thing (line 4-7). In the iteration I filter out some particular users.

And this is the interesting part. The twin collection also tracks which users have been added since we created the form. It allows me to compare the currently iterated user and the `added` array which contains users that have been newly added, only (line 5). Using this technique, I can really limit this loop to recently added user objects. I access the nested user form's `model` which is of course a `User` instance. The associated authorship's `confirmed` flag gets zeroed, and everything is perfect (line 6).

We will encounter a few more handy tracking methods the twin provides for us in this chapter. Regardless of the details, let's quickly think about what we has changed.

Imperative Callbacks

In Rails callbacks happen at well-defined events. The ActiveRecord life-cycle defines those events, like `before_validation` or `after_save` and once you add callbacks to the hooks, it is beyond your control when they are run. Whenever the model thinks it is "after save" time it will run all the attached callbacks for that hook, regardless of whether you want that or not.

In Trailblazer, this changes substantially.

We completely skip defining life-cycle events. If you need to run callbacks before or after a certain point you do this by explicitly triggering hooks or simple methods in your operation. For example, a `before_validation` hook literally happens *before* the validation.

```

1 def process(params)
2   comments_count_ok? # before validate.
3   validate do
4     # ...
5 end

```

Instead of the implicit callback invocation as found in ActiveRecord, in Trailblazer you instruct hooks to run their callbacks at your will. I call this *Imperative Callbacks* in Trailblazer.

Twins help identifying and tracking events. The intermediate twin object graph collects low-level events like adding, deleting, destroying, creating and updating. Since twins are mostly trees of objects, you can run callbacks for a deeply nested structure without losing control of what to trigger when.

We already learned how that could look like in the chapter at hand. Basically, we asked the twin for occurrences of a particular event like “*item added to collection*” and then ran the callback for all matching twins.

```

1 do_this! if contract.users.added.include?(user)

```

Explicitly invoking callbacks does not only remove magical surprising triggers from your code, they also make it more readable. Well, I am not saying the above code is particularly readable, but when looking at operational code you can instantly see which hooks and callbacks are involved.

Speaking of readability, Trailblazer comes with a mechanism built-in to handle callback invocations. This abstracts dispatch to a very clear and concise declaration.

```

1 class Thing::Create < Trailblazer::Operation
2   callback do
3     collection :users do
4       on_add :notify_author!
5       on_add :reset_authorship!
6     end
7
8     on_create :expire_cache!
9   end

```

This is pretty straight-forward and I’m not sure if there even is need to explain since you already understand what this will do. Anyway, you paid for this book so let’s go through it.

Operations allow you to structure sets of callbacks into *groups*. The `callback` block opens a new `Disposable::Callback::Group` class for you and provides the DSL to configure the callbacks (line 2-9). Usually, an operation will only have one group, but later we will learn that you can have as many callback sets as you desire.

Note that this has nothing to do with your contract, even though we're using the same API. In Trailblazer, we heavily rely on schemas and reuse API to define nested graphs.

Before we walk step-by-step through the chain of events that happen here, let me show you how this callback group is invoked in your operation.

```
1 def process(params)
2   validate(params) do |f|
3     f.save
4     dispatch!
5   end
6 end
```

By calling `dispatch!` you instruct Trailblazer to run all callbacks from the default group (line 4). This means, the callbacks will be invoked at this very moment of the execution, right after saving the object graph. It is up to you where exactly you dispatch your callbacks. Also, keep in mind that you can exclude particular callbacks from the `dispatch!` call and maintain several different sets of callbacks. We'll speak about that later when we need it.

Here's the callback group, again.

```
1 callback do
2   collection :users do
3     on_add :notify_author!
4     on_add :reset_authorship!
5   end
6
7   on_create :expire_cache!
8 end
```

When calling `dispatch!` the following callbacks will be invoked. Note that order matters here.

1. The callback mechanism will query the contract for items having been added to the `users` collection using the tracking logic we learned earlier. For each item the `notify_author!` method is called.
2. Same happens again. This time, the `reset_authorship!` method gets invoked per added user.
3. Since the top-level object, the `Thing` instance we just persisted, is considered "created", the `expire_cache!` method is run after that.

Callbacks in Operations

Grouping callback logic makes it incredibly simple to structure complex post-processing code in operations.

There are two ways of defining callbacks. You can either use *inline callbacks*, specify callbacks and methods in the operation class. Or, and that's my preferred way, maintain separate callback objects and their methods cleanly encapsulated in a different location.

Inline Callbacks

Using inline callbacks in an operation means you define the event callbacks in the operation class.

```
1 class Thing::Create < Trailblazer::Operation
2   callback do
3     collection :users do
4       on_add :notify_author!
```

Inline callbacks require you to not only specify the callbacks in the class, but also the implementing method that represents the callback code.

Have a look at our first callback.

```
1 class Thing::Create < Trailblazer::Operation
2   # ...
3   def notify_author!(user, options)
4     return UserMailer.welcome_and_added(user, model) if user.created?
5     UserMailer.thing_added(user, model)
6   end
```

As noted earlier, the `notify_author!` method here sits in the operation. It receives the nested form that matches the event `on_add` in the `users` collection (line 3). By using the twin API we discovered in the beginning of this chapter I check if the user was just created, and, if yes, dispatch further to an imaginary mailer (line 4). Note that I have access to both the nested form (or twin) that was added and the `Thing` instance that I can access via `Operation#model`.

The options hash allows to inject arbitrary parameters into callback methods. We'll learn about that more in a minute, when discussing callback objects.

The second callback `reset_authorship!` is almost as boring as the mailer one we just wrote.

```
1 def reset_authorship!(user, options)
2   user.model.
3     authorships.find_by(thing_id: model.id).update_attribute(:confirmed, 0)
4 end
```

This code is a bit awkward and probably a result of my lack of understanding for ActiveRecord. Since the nested user form is passed into the method, I access the actual User instance via `user.model` (line 2). I then find the particular authorship model that represents the binding between the user and the thing, and set this `confirmed` flag to zero (line 3).

We also had third callback `expire_cache!` in the group. I left it there to point out the flexibility of callback groups, but please let me put you off until we discuss view caching in a minute.

The callback group has made is really simple and intuitive to add post-processing logic to our operation. The familiar nesting DSL lets you add callbacks to nested members of the object graph, even to scalar properties, which I'm gonna demonstrate in a few pages.

Inheriting Groups

As you do remember, the `Update` operation inherits from the `Create` operation. This means it will also inherit the callback groups you defined. It is safe to inherit the `on_add` callbacks, as they will only get triggered when a user got added. This is the behavior we want in both create and update operations.

In our update context, the `on_create` won't be run, though, as we don't create but update a `Thing` instance. This will result in the `expire_cache!` method never being called.

Luckily, we can add and remove properties in subclassed callback groups, just as we do in contracts, twins, and representers.

```
1 class Thing::Update < Trailblazer::Operation
2   callback do
3     on_update :expire_cache!
4   end
```

The expiry method will now be called on updates, too, but only in `Update` context (line 3). However, we can simplify both operations by using the `on_change` event which handles `create` and `update`, Resulting in the `Update` operation not having to define its own group.

```

1 class Thing::Create < Trailblazer::Operation
2   callback do
3     # ...
4     on_change :expire_cache!
5   end

```

This will surely not work for every case. Groups allow you to fine-tune inherited definitions or to completely override them. We will come across this several times in this book³⁹.

Callback Object

While inline callbacks are a nice way to quickly define post-processing logic, they blow up the operation class pretty fast. Trailblazer also allows referencing external callback objects, so you can implement them in a separate class and file.

Since this is my preferred approach, I introduce a new file app/concepts/thing/callback.rb with the following content.

```

1 module Thing::Callback
2   class Default < Disposable::Callback::Group
3     collection :users do
4       on_add :notify_author!
5       on_add :reset_authorship!
6     end
7
8     on_change :expire_cache! # on_change
9
10    def notify_author!(user, options)
11      # ...
12    end
13
14    def reset_authorship!(user, operation:, **)
15      user.model.authorships.find_by(thing_id: operation.model.id).
16        update_attribute(:confirmed, 0)
17    end
18  end
19 end

```

As you can see, this is a completely isolated class with no inherited dependencies. I put the new class in the Thing::Callback namespace, similar to what we do with contract and representers soon (line

³⁹If you're curious now, learn more from the [API docs](#).

1). The callback object is derived from `Disposable::Callback::Group` (line 2). To define the event and handlers, I use the same API as we did in the inline callbacks earlier (line 3-8).

What is different now is that the implementation of the methods do no longer have to sit in the operation, but in this very class (line 10-17).

We have to change our implementation slightly, as this object doesn't have access to the operation and model directly. When using callback objects, Trailblazer will automatically pass in operation, params and contract into callback methods (line 14). I use keyword arguments to filter out the `:operation` parameter and grab the model via `operation.model` (line 15).

To register the new object with an operation, we need to go back into `app/concepts/thing/operation.rb`.

```
1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     callback :default, Callback::Default
```

Instead of defining the callbacks using a block, we simply pass the class name to `callback` (line 3). This will resolve to `Thing::Callback::Default` and reference the callback object we just implemented.

Testing Callbacks

Before we move on, let's make sure real quick that we write a few tests for those callbacks. There's different ways to achieve this. In this section, I'm gonna speak about the quick way, only.

Callback tests go into the operation test, too. in `test/concepts/thing/operation_test.rb`. While you can extract them to separate blocks I will include them into existing test cases. It's better to test "too much" in one case than to forget about things and not test them at all.

```
1 class ThingOperationTest < MiniTest::Spec
2   it do
3     # ...
4     # authorship is not confirmed, yet.
5     model.authorships.pluck(:confirmed).must_equal [0, 0]
6
7     op.invocations[:default].invocations[0].must_equal
8       [:on_add, :notify_author!, [op.contract.users[0], op.contract.users[1]]]
9   end
```

In the test case we coded a while ago, a valid operation is run that adds two users. The test already covers the authorship reset and makes sure both `confirmed` flags are zeroed (line 5). This is a

functional test that really checks the effect on the application state and act as an example for good callback tests. With this case, the `:reset_authorship!` callback is covered.

Since we're not finished with implementing our operation I added a basic test for the `:notify_author!` callback. The operation tracks each callback execution in the `invocations` field. By making sure the cryptical tracking object contains the right values, `:notify_author!` combined with `:on_add` and the two user twins we just added, we have written a quick and dirty test (line 7-8).

We will soon come back to this test case and make it look and feel appropriate. This is really just to show you how you can assert callback invocations without actually testing the execution impact.

In the next chapters I'm gonna make you write a few tests for the callbacks. We could test them manually, but callback behavior should always be assessed in combination with running the entire operation.

Another question I often hear is "*Shouldn't we be able to turn off callbacks in tests, to make them faster?*". The answer is: No. You will end up with a half-baked test environment that soon will diverge significantly from the real one.

Most callbacks have side-effects, like setting flags (`mail_sent`) or running a small piece of logic somewhere that changes application state. Skipping callbacks in tests will create a fake world and things might pass even though they are broken in production - something I have seen in many Rails apps that insisted on a "fast" test suite by testing something that is never gonna happen that way.

My personal preference is: Always run the complete operation, with all its callbacks. My test environment will be 100% identical to production and bugs are easier to spot.

If you still think you have to turn callbacks off check out the docs to learn how to do that.

So far, the `Thing::Create` operation consumes the incoming form, validates it and saves the new object graph to the database. Likewise, the `Update` handles the editing. After that is done, a number of callbacks are triggered. Let us now move to a topic that might seem completely unrelated to what we've done in this chapter. We will now learn about view caching and how we can use callbacks to expire caches.

View Caching

Let's approach view caching step-wise. Before we implement the caching for the things grid, I want to introduce you to this technique by caching only the `show` state of the `Thing::Cell`. You remember, that's the cell that only renders one thing's name and creation date.

Caching in cells exclusively works with what I call *state caching*. This means, you define on the class layer which state you want to cache. As most cells only have one public state, this will often result in a snippet similar to the following.

```

1 class Thing::Cell < Cell::Concept
2   cache :show
3
4   def show
5     render
6   end
7 end

```

The first argument passed to `cache` is the state that needs caching (line 1). You could also provide instructions for computing a cache key here, and we'll do that in a minute, but allow me to discuss what's going to happen now⁴⁰.

Suppose this cell gets rendered twice by invoking our old helper friend `concept("thing/cell", thing).()`. This could happen in two different requests, or two subsequent calls in the same view. Here's what Cells will do.

1. In the first cell invocation, the `show` state will be run as always. However, since Cells knows we enabled state caching for this state, the return value of `show` will be saved in the cache store. Usually, this is a HTML fragment rendered by your cell being pushed to the Rails cache layer.
2. Next time the cell is rendered, it will look into that cache store if there's a cached version of the `show` state. And since we rendered the cell before, the cell will find a fragment in the cache store, and return the identical HTML fragment without actually running `show` again.

While this will greatly speed up your cell rendering, you will lose all dynamics. Your cell state is cached forever since we do not provide a cache key. Regardless of the input to the cell, the cache result will always be identical.

In order to maintain different cached versions, we need to provide a cache key. I call this *versioner* but that's just my lingo.

Cache Keys

Cells comes with different options to compute cache keys. You could simply pass an expiry time to the cache store.

```
1 cache :show, expires_in: 10.minutes
```

However, this will still mess up your view by providing one and the same fragment for possibly different inputs to the same cell. My favorite versioner style is passing a block.

⁴⁰You might be wondering why caching happens in the class and not in the view, as Rails does it. Fragment caching is [not implemented in Cells per design](#) - Cells enforces an object-oriented design rather than cluttering your views with caching blocks.

```
1 cache :show do
2   [model.id, model.updated_at]
3 end
```

Whatever you return from that block Rails will try to convert to a cache key. Arrays work well as they will be joined to one string by the underlying engine.

The block is executed in cell instance context before the actual state is invoked. That's why I can access `model` and use both the `id` and `updated_at` field to compute a unique cache key for this cell (line 2).

Now, imagine this block being called whenever the cell is invoked from a view. It will always compute a different cache key for a different model that you pass into it. However, if you pass in the same model, the key will only differ when the model's `updated_at` field has changed in the meantime.

For a better understanding, here's an example session.

```
1 thing #=> <Thing id: 1 updated_at: 22/06/2015-19:38>
2
3 concept("thing/cell", thing).()
4 #=> "thing/cell/show/1/22/06/2015/19/39"
```

The last line shows the cache key this cell will use to store the rendered view (line 4). It will render `show` and push the fragment to the cache store using the key it computed before running `show`.

If you repeat this call, maybe in the next request, and the `thing` instance still has the same attributes, the cell will calculate the same cache key, find the existing fragment in the store, and return that instead of re-running `show`.

Expiring Keys

Now, assuming the `thing` got edited a while ago, maybe you changed the description, this will also change the `updated_at` attribute. When re-rendering the cell with the same `thing` this will result in a new cache key.

```
1 thing #=> <Thing id: 1 updated_at: 01/01/2016-12:09>
2
3 concept("thing/cell", thing).()
4 #=> "thing/cell/show/1/01/01/2016/12/09"
```

Even though the model is still the same, this is a completely different key, and this is desired! Since the `thing` got updated, you want the cache to expire and re-render, as crucial data like the name or

the description might have changed. When invoking `show`, the cell won't find a fragment for the new key in the store, yet, and re-run the state.

By cleverly picking a cache key algorithm, you save yourself from having to manually delete the old cache entry from the store. The cache store will automatically dispose of the old entry after detecting it hasn't been used for a certain while.

Debugging View Caching

We're almost set to implement the compound grid caching now. Before we start working on cache key and expiry, here's another cool trick to debug your caching logic. Caching is a nebulous and fuzzy affair and it's always good to see what is going on behind the scenes.

First of all, when working on caching I always turn on caching in `config/development.rb`.

```
1 Rails.application.configure do
2   config.action_controller.perform_caching = true
```

Enabling this will activate caching logic in Cells, too. Under normal development conditions, caching is turned off and your versioners will never be run. It is a good idea to thoroughly test cache logic before pushing it live.

Cells comes with caching notifications that integrate with Rails notification mechanism. This has to be manually included into the cells you want to monitor.

```
1 class Thing::Cell < Cell::Concept
2   include Cell::Caching::Notifications
```

By including the `Notifications` module cache reads and writes from the cell will be reported (line 2).

As a third step, you need to subscribe to these specific notifications. In Gemgem, I added a new file `config/initializers/cells.rb` that includes the following subscription assignments.

```
1 ActiveSupport::Notifications.subscribe "read_fragment.cells" do |name . .
2   Rails.logger.debug "CACHE: #{payload}"
3 end
```

From now on, whenever caching-related cells logic is run, you will see output similar to this snippet in your server log.

```

1 CACHE write: { :key=>"cells/thing/show/39/26/5/6/19/6/2015/5/170/false/UTC" }
2 CACHE: { :key=>"cells/thing/show/39/26/5/6/19/6/2015/5/170/false/UTC" }
```

An extremely helpful tool to observe the caching logic. A cache write log is the opposite of a *cache hit* - it means the fragment with the mentioned cache key wasn't found⁴¹.

Caching Composed Views

We discussed how to cache single boxes of our thing grid on the home page, where every thing box keeps its own cache fragment. This is a valid option if Gemgem was a high-frequency page where hundreds of things get added every minute and the front page always has to be up-to-date.

Gemgem is just starting to roll, though. To minimize rendering on the home page we should simply cache the entire grid and expire its cache whenever a new thing got added or an existing one was updated.

The Grid cell has its own show state. Equipped with the right versioner, we will be able to cache the entire grid without having to implement caching for each box. This in turn means we can remove the cache declaration from Thing::Cell and instead define the cache on Thing::Cell::Grid.

```

1 class Thing::Cell < Cell::Concept
2   class Grid < Cell::Concept
3     cache :show do
4       Thing.latest.last.id
5     end
6
7     def show
8       things = Thing.latest
9       concept("thing/cell", collection: things, last: things.last)
10    end
11  end
```

This is an example of a cache versioner for the grid. Since the `show` method grabs all things to display by using `Thing.latest`, we could use the newest model's id as cache key (line 4). Our cached grid fragment would remain cached until we add a new `Thing`.

A very simple solution, with one problem, whatsoever. The cache wouldn't expire when an older thing was updated as this won't change the latest thing's id.

⁴¹By the way, the Cells caching notifications are a result of community effort and came from a [pull request](#). I didn't even know that notifications existed back then and now I'm glad someone else solved this for me in Cells.

The CacheVersion Pattern

When it comes to complex composed views and caching I often use a technique I named *CacheVersion*. I am pretty sure there's at least two dozens patterns from official geniuses out there with the same behavior but a cooler name, anyway, this pattern has saved millions of rendering cycles in our production apps.

For example, I applied this to a comment forum where each comment is rendered from a cell, and the entire thread is embraced by a container cell, that can postload pages of comments when paginating.

The concept of CacheVersion is - as always - extremely simple and I hope you will not be disappointed by such an obviously trivial solution: Use a persistent store to maintain a cache key and deliberately update that cache key when the application state changes.

Applying CacheVersion combines the simplicity of cache keys with the reliability of expiry. What am I talking about? I have no idea, but let's walk through it and we both will understand.

```
1 class Grid < Cell::Concept
2   cache :show do
3     CacheVersion.for("thing/cell/grid")
4   end
```

The cell's versioner doesn't implement its own key but dispatches to a CacheVersion object. Note that I pass a token "thing/cell/grid" to that remote versioner. The token could be anything and will be the reference of that particular cache version (line 3). It is completely irrelevant what exactly this version represents, the important concept is that the cell knows the name. Basically, the cell asks "*Hey, CacheVersion, what's the actual version of XYZ?*".

Here is the table layout of cache_versions.

```
1 create_table "cache_versions" do |t|
2   t.string   "name"
3   t.datetime "updated_at"
4 end
```

The CacheVersion implementation will then try to find the version of XYZ using a persistent store. I use ActiveRecord here, but this could be implemented using the much faster Redis store or anything you like. The class goes into app/models/cache_version.rb.

```

1 class CacheVersion < ActiveRecord::Base
2   def self.for(name)
3     where(name: name).first or create(name: name)
4   end
5
6   def cache_key # called in AS::Cache#retrieve_cache_key.
7     updated_at
8   end

```

The `for` method will either find the version entry for `XYZ` or create and persist a new one (line 3). And now the trick: Our cell versioner returns this `CacheVersion` object to Rails' caching layer in order to compute a cache key.

Rails will try to call `cache_key` on this object and succeed because we implemented that instance method (line 6-8). The method will return the `updated_at` string, a field automatically maintained by Rails. This field value is our cache key for the token `XYZ`.

As long as this `updated_at` field doesn't change, the cache key for `XYZ` will always be identical resulting in our grid cell being cached.

The `CacheVersion` class defines one more method to expire caches. Note that this is an instance method and in order to expire you need to hold the respective version object in your hands.

```

1 class CacheVersion < ActiveRecord::Base
2   # ...
3   def expire!
4     update_attribute(:updated_at, (updated_at || Time.now) + 1.second)
5   end
6 end

```

The `expire!` method allows you to change the cache key for `XYZ`. It will simply increment the `updated_at` field by one. The next time the versioner asks for the cache key it will be a different key and result in re-rendering of the grid.

To wrap that up.

1. The cell's versioner queries the `CacheVersion` class for "its" key. It has no knowledge other than "its" cache name.
2. The `CacheVersion` finds or creates a key. The key is simply the `updated_at` field but could be anything. This key is persisted as a row in the `cache_versions` table.
3. As long as the key wasn't changed, the cell will be cached.
4. Expiring the cache key happens whenever the grid changes. So, expiry logic goes into operations.

Expiry in Operations

The thing grid only needs to be updated when a new thing got added and takes the first place in the grid. And, don't you forget about that, when a thing was updated. The changes made have to be reflected in the grid⁴².

Both creating and updating things only works via operations, so the `Thing::Create` and `Thing::Update` classes are where we trigger cache expiry.

You remember that we already had a callback in place in the `Create` operation, right?

```
1 class Thing::Create < Trailblazer::Operation
2   callback :default, Callback::Default
```

We now need to add the expire callback to the `Default` group in `app/concepts/thing/callback.rb`.

```
1 module Thing::Callback
2   class Default < Disposable::Callback::Group
3     on_change :expire_cache!
4
5     def expire_cache!(thing, *)
6       CacheVersion.for("thing/cell/grid").expire!
7     end
```

Both the registration and implementation are pushed into the group (line 3-7).

The callback method `expire_cache!` grabs the `EventVersion` instance representing the grid state and calls the aforementioned `expire!` method, which will increment the cache key (line 5-7). As a consequence, the next time this cache key is retrieved from the cell, it won't find a cache entry and re-render the grid.

Discussion

A few things I have to criticize about this approach.

First, didn't we say logic shouldn't be in models? Why am I putting expiry and caching logic into the `CacheVersion` class? This is surely breaking Trailblazer's idea of logic-less models. At a later point, we will move it to an operation, so we can trigger expires from the console, too.

Does the operation really need to know about view details? Why does the `expire_cache!` method pass the cell's view token to `CacheVersion`? That is tightly coupling business and view. The answer: It's a quick and easy way to achieve what we need now. When caching gets more complex, the

⁴²For completeness, I have to mention that I modified `Thing::Cell` slightly: I added the `description` property to the view. When a user edits a thing and changes the description this requires the grid to update. Both `implementation` and `tests` can be found in the Gemgem repository.

operation could dispatch to a cache expire class that knows what actions affects which cached parts of the app.

Nevertheless, I have used this technique successfully in many applications. The point of CacheVersion is to minimize persistence access in order to compute a cache key. Instead of trying to generate an MD5 hash from all Thing instances visible in the grid, as it is often done, we only need exactly one database lookup to find, or compute, the cache key.

File Uploads

The application now properly notifies authors, sets up the desired environment when adding and updating things, and expires caches. To close this chapter I'd love to add a file upload. We could allow users to upload an image for things to make the things pages a bit more colorful.

As a first step, we should add a file upload field to the thing form in `app/views/things/new.html.haml`.

```
1 = simple_form_for @form do |f|
2   = f.input :description
3   = f.input :file, as: :file
```

The form object property I call `file`, and I need to provide the type to SimpleForm as the type of this virtual field can't be inferred (line 3).

Of course, we need to add the `file` property to the form now.

```
1 class Thing::Create < Trailblazer::Operation
2   contract do
3     # ...
4     property :file, virtual: true
```

This property needs to be virtual as the underlying model `Thing` doesn't have such a field (line 4). We will need a little bit of more code to make this work. However, when submitting the form with a selected file to upload, the form's `file` property will now contain the uploaded file.

```
1 def process(params)
2   validate(params) do |f|
3     raise f.file.inspect #=> <UploadedFile>
4   end
```

Nothing else happens so far. The form won't try to set `file` on the model as the property is declared virtual. The form simply keeps a reference to the uploaded file object, nothing more.

Paperdragon

I want to store the original and a thumbnail version of the image. In order to process the upload we need a file uploading gem. In Gemgem, I use [Paperdragon](#)⁴³, but this works fine with CarrierWave, too.

I chose Paperdragon because it has absolutely no coupling to ActiveRecord. The entire processing is in your hands. While this is a tiny little bit more verbose, you fully control when image processing, S3 syncs, and so on happen.

We could use Paperdragon's manual API to process and store the images, or the `Model` module which gives us a fuzzy API to upload and render images.

I will go the explicit way, use the `Model::Writer` feature in our contract which introduces one new method to process a file.

```

1 contract do
2   # ...
3   property :file, virtual: true
4
5   extend Paperdragon::Model::Writer
6   processable_writer :image
7   property :image_meta_data

```

After including that model, I instruct Paperdragon to provide me a single method `image!` for the processor instance. This happens by calling `processable_writer :image` (line 6).⁴⁴

The only requirement Paperdragon has to the class it is operating on is a writeable field named `image_meta_data`. After processing the different versions and storing the images, Paperdragon will push locations of images, file names, etc. to that field. As this field needs to get persisted, I added a same-named TEXT column to things.

```

1 class Thing < ActiveRecord::Base
2   serialize :image_meta_data

```

This field will contain a serialized Ruby hash, so I instruct Rails to automatically handle the serialization and deserialization on the database level by using `serialize` (line 2).

Image Processing

After the infrastructure has been set up, let's implement the processing. In the first version, I will simply call the upload code directly in the operation's `process` method without using a callback.

⁴³<https://github.com/apotonick/paperdragon>

⁴⁴Including the `Writer` into the contract might feel messy. You can use a separate uploader object for the actual processing. The minimal code change [is here](#).

```

1 def process(params)
2   validate(params[:thing]) do |f|
3     upload_image!(f)
4     # ...
5 end

```

The `upload_image!` method in the operation uses Paperdragon's API to create and store the two versions of the uploaded file.

```

1 def upload_image!(contract)
2   contract.image!(contract.file) do |v|
3     v.process!(:original)
4     v.process!(:thumb) { |job| job.thumb!("120x120#") }
5   end
6 end

```

Calling the `image!` method that Paperdragon provides on the contract and passing the actual file into it will instantiate a Paperdragon processor and allow operations on the file (line 2). Note that it's your job to provide the `file` object, Paperdragon has no idea that the processor in `image!` and the `file` field from the form are related.

In the block, I use Paperdragon's API and instruct it to store an unprocessed version of the file and store it as the `:original` (line 3). The `:thumb` version I crop and resize to 120x120 pixel (line 4). Paperdragon uses the Dragonfly gem under the hood and supports all modifications, conversions, etc. Dragonfly allows.

After that block is run and the images are processed, Paperdragon will store them in the configured location⁴⁵. It will also create a meta data hash with file names and push that to the contract. Paperdragon does this by simply calling `contract.image_meta_data= {}` after the block.

If you're curious, you can have a look at this meta data yourself.

```

1 validate(params[:thing]) do |f|
2   upload_image!(f)
3   raise f.image_meta_data.inspect

```

So far, the image got uploaded, processed, stored and the meta data got written to the contract's `image_meta_data` field. How is that persisted?

We already learned that the form twin will push all defined properties to the underlying model when we call `sync`. The `image_meta_data` field is a property, and therefore will be written to the model just like any other field on the form.

⁴⁵You need to configure where Paperdragon should store files. This happens in a separate initializer `dragonfly.rb`.

File Validations

Two security holes need fixing before moving on. Firstly, the uploaded file can be just anything. I install the excellent `file_validators` gem⁴⁶ in order to get generic file validations into my form. A huge benefit of this gem is that it doesn't have any code relating to ActiveRecord, which would make our form break.

```
1 contract do
2   property :file, virtual: true
3   validates :file, file_size: { less_than: 1.megabyte },
4     file_content_type: { allow: ['image/jpeg', 'image/png'] }
```

The new `:file_size` and `:file_content_type` option for `validates` help us excluding unwanted formats and files too big (line 3-4). We will write tests for that shortly.

Another problem is that `image_meta_data` is writeable in the form. This means someone could inject this field into the `params` and thus write arbitrary content into our thing's `image_meta_data` attribute. This is not a problem when we do upload a file: the field gets overridden in Paperdragon, anyway. However, this will allow script kiddies to change that field when we do not upload but simply update a field of the form.

We can't set it `writeable: false`, though. This would result in this field not being synced to the model when we `save`. The property should only be write-protected when deserializing the incoming `params` hash and populating the form.

```
1 property :image_meta_data, deserializer: {writeable: false}
```

This can be achieved by using the `:deserializer` option. I won't go into detail here, but basically this option is only applied to the deserializer. Setting properties to `writeable: false` here won't write the incoming value to the form in `validate`.

Uploading: An Overview

You might be irritated about this because back in the days, with Paperclip, everything worked out of the box without having to do anything manually. Also, back in the days, when things went wrong or you wanted to change a particular processing step, you found yourself standing in front of a black box that magically hooks into arbitrary life-cycle events of your model.

It is incredibly hard to control uploads with Paperclip. In another project, we had to fine-tune processing and storing in S3, we needed control over the creation and pushing of every file version. This eventually resulted in Paperdragon, which uploads thousands of images a day in a photo community now.

⁴⁶https://github.com/musaffa/file_validators

Paperdragon in combination with Reform and Trailblazer's operation makes it really explicit what is going on when, and allows you to change flows accordingly. It makes it quite easy to add debugging or strategical prys in your code to find out more. Here's a quick wrap up of the process.

1. You submit a form with an image file.
2. After `Form#validate`, the `file` property will now point to this file object.
3. When validations are run, file-specific checks are run against the `file` property. No processing or storing has happened, yet.
4. In the operation that called `validate`, Paperdragon is invoked via the `image!(file)` accessor of the form and allows us to process and store different versions of the images. This is not just a nice DSL but really executes the commands in real-time, providing us an interface to the image operations.
5. With completion of the processing and storing, a meta data hash is compiled and pushed onto the form's `image_meta_data` field.
6. Finally, when the form gets saved the `image_meta_data` field is written to the model that now persists where the images can be found in the file system.

Again, the model isn't touched until we save the form.

Callback Groups

When clicking through the thing forms, you will soon notice that there's an exception raised when you did not add a file to upload. This is because we always run the upload logic in process without checking whether there's a file present or not.

I won't bother you with different ways to solve this. Of course you could check whether the params hash contains a file object, as I've seen it a lot in Rails hacks, but we can simply use a twin feature we learned earlier.

```
1 def process(params)
2   validate(params[:thing]) do |f|
3     upload_image!(f) if f.changed?(:file)
4     #...
5   end
```

To avoid running the upload logic without an actual file to process, I use the `changed?` method of the form twin (line 3). This will run `upload_image!` only when "something" has been assigned via `file=` on the form, which is exactly what happens in `validate` but only when a file is submitted.

Another option is to introduce a second callback group. Trailblazer allows you to have as many groups as you need. Here's how we could restructure the callback code in `app/concepts/thing/-callback.rb`.

```

1 module Thing::Callback
2   class BeforeSave < Disposable::Callback::Group
3     on_change :upload_image!, property: :file
4
5     def upload_image!(thing, *)
6   end

```

We now have two callback groups Default and BeforeSave (line 2). I use the `on_change` event and limit it to `file` using the `:property` option (line 3). Yes, you can apply events to single properties, too. As always in callback groups, the `upload_image!` method resides in this class, too (line 5).

To trigger that new callback, it needs to get registered in the operation.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     #..
4     callback :before_save, Callback::BeforeSave

```

I alias the group to `:before_save` to indicate where it's called. We could also name it `:upload`, but we will reuse this group for more than file works in the next chapter.

The way I dispatch the two groups is conceptually identical to before and after save hooks.

```

1 def process(params)
2   validate(params[:thing]) do |f|
3     dispatch!(:before_save)
4     f.save
5     dispatch!
6   end
7 end

```

Calling the new `:before_save` group happens where we originally had the manual invocation of `upload_image!` before saving the form. As you can see, `dispatch!` also accepts a group name to run an arbitrary callback group (line 3).

Personally, I love separate callback groups even when they contain one single event, only. Quickly you develop your style of groups you have in every operation. What I appreciate here is the freedom to choose how many groups you need and where to trigger them.

Rendering Images

After all this uploading work, we deserve to see a visual result. Luckily, rendering uploaded images is a walk in the park with Paperdragon. I will add a new cell named `Thing::Cell::Decorator` that will keep several public decoration methods. This is a practice I often use when I need a generic decorator cell for a concept.

```

1 class Thing::Cell::Decorator < Cell::Concept
2   extend Paperdragon::Model::Reader
3   processable_reader :image
4   property :image_meta_data
5
6   def thumb
7     image_tag image[:thumb].url if image.exists?
8   end
9 end

```

This time, I include the `Paperdragon::Model::Reader` module that will give me a plain reader. I have to configure the reader's name via `processable_reader :image` (line 3). That reader from `Paperdragon` will call `image_meta_data`, so I have to provide that field, too.

Again, `image_meta_data` is the only requirement `Paperdragon` has to the using class. Therefore, I need to delegate this method to the model using `property` (line 4).

The `thumb` method is an object-oriented version of what we used to call *helper*.

The `exists?` method allows to check if there really is an image uploaded, and by using the `[:thumb]` reader, we get the attachment object for the thumb version (line 7).

Not too much happens here in the background. `Paperdragon` will open the `image_meta_data` hash of the model, grab the `:thumb` image path and return it. And voilà - here comes a beautiful, cropped, resized and slightly sharpened version of whatever photo you uploaded.

In `app/views/things/show.html.haml` we can now render the thumb of the uploaded file by using our decorator cell.

```

1 %h1
2   = concept("thing/cell/decorator", @thing).(:thumb)

```

Note how I call the state explicitly here. `cell.(:thumb)` will invoke the `thumb` method and return the image tag. Nicely encapsulated, the user doesn't need to know any details about `Paperdragon` details.

Have another look at the rendered image in your browser, and enjoy the beauty for another blink of an eye. Then, it's testing time!

Testing Uploads

Luckily, we're using a well-tested gem for the processing and save testing the implications coming with it. What we need to make sure is that an upload actually works with a valid file. An upload test is just another operation test and goes into `test/concepts/thing/operation_test.rb`.

```
1 it do
2   thing = Thing::Create.(thing: {name: "Rails",
3     file: File.open("test/images/cells.jpg")}).model
4
5   Paperdragon::Attachment.new(thing.image_meta_data).exists?.must_equal true
6 end
```

The test file is literally a `File` instance that I open from my fixtures (line 3). I pass that file object into the operation. This is the analogue to what happens in a real form submission with a file upload. Rails will provide a `File` instance in params.

The `Create` operation will handle all the internals and all I need to do is check the end result by making sure the image version was really processed and stored. I do this by using `Paperdragon`'s `Attachment` class, pass in the `image_meta_data` and let it check for its existence (line 5).

We will soon learn how to replace this slightly clumsy test with a view model invocation.

The opposite test case is even simpler.

```
1 it "invalid upload" do
2   res, op = Thing::Create.run(thing: {name: "Rails",
3     file: File.open("test/images/hack.pdf")})
4
5   res.must_equal false
6   op.errors.to_s.must_equal "{:file=>[\"file has invalid extension\"]}"
7 end
```

Here, I pass a PDF file into the operation with the evil, terrifying name "`hack.pdf`" (line 2-3). Validation result should be wrong and the error message generated by `file_validations` should be present (line 5-6).

Decorator Test

It is also a good idea to test the decorator cell we wrote. This test goes into `test/concepts/thing/-cell_test.rb` and is as simple as it could be.

```
1 describe "Cell::Decorator" do
2   it do
3     thing = Thing::Create.(thing: {name: "Rails",
4       file: File.open("test/images/cells.jpg")}).model
5
6     concept("thing/cell/decorator", thing).thumb.must_equal "<img src=...>"
7   end
8 end
```

More and more it becomes obvious that replacing factories with operations is the way to a clean, realistic test application state. The Create call with a valid file is how I set up the test thing (line 3-4). Note that you're free to extract this kind of code into factories. As long as they internally use operations this works just fine.

As a last step I invoke the thumb method on the cell and test if the returned img tag is what I expect it to be (line 6). An interesting point here is that I did not use the call style, but call the method directly. This is a trick to avoid the content being wrapped in a Capybara string and allows me to test for equality here.

While this is totally fine in tests, in real views this will also bypass caching, so please call methods via call in rendering environments.

I'll spare you the details of the second, negative test which makes sure no image tag is rendered when there's no uploaded file.

Conclusion

Callbacks as found in Trailblazer are different to what we've learned in Rails. Callback groups might seem a bit more verbose when you start using them but soon it becomes visible how that little bit of extra code makes them so much cleaner, predictable and also reusable.

Instead of hooking them directly into the persistence layer, callbacks are separate classes that operate on the twin API to find out when they apply.

Even though we define and talk about events, please don't confuse Trailblazer's callback system with a full-blown event dispatcher that magically triggers callbacks in real-time. Imperative callbacks are passive, you say when you need them to be run.

Also, callback groups are just a simple abstraction. Further dispatching has to happen in a different layer. For example, I would never implement a complex notification system with dependencies, pushing, etc. directly in callbacks. The operation will identify events, pick that up in a callback and can then delegate that to a notifications class.

For now, let's go and discover the crazy world of authentication!

Authentication

We have grown up and it is time to establish some policies and rules in Gemgem. The next two chapters will talk about authentication and authorization, topics relevant for every web application.

Users should be able to sign in and out, register themselves for Gemgem and have a different experience when signed in. Also, implicitly created users, for instance when commenting, should be able confirm their account in a separate workflow.

The branch for this chapter can be found [in the repository](#)⁴⁷.

Populating by ID

So far, when adding comments on the thing page it is a requirement to add the user's email. After submission, besides creating a new comment row, this will also build a brand-new user object, regardless of whether or not the email belongs to an existing user of Gemgem.

The first step for a coherent user integrity is to create new users only when the comment author's email is unknown. If that's not the case, the existing user should be associated with the new comment. Of course, we need to find this existing user.

In order to do so, we need to replace the comment's `Create#setup_model!` method, where we statically added the `User` instance to a comment. We substitute it with a combination of a prepopulator and a populator.

```
1 class Comment::Create < Trailblazer::Operation
2   contract do
3     property :user,
4       prepopulator: ->(*) { self.user = User.new },
5       populator: :populate_user! do
6         # .. nested setup ..
7       end
8
9     def populate_user!(fragment:, **)
10      self.user = (User.find_by(email: fragment["email"]) or User.new)
11    end
12  end
```

⁴⁷<https://github.com/apotonick/gemgem-trbrb/tree/chapter-09>

The `:prepopulator` makes sure that, once we call `prepopulate!`, the comment form will always contain a nested `User` instance (line 4). When processing the submitted form in `validate`, the `:populator` option allows us to hook in our own code to populate the form. Instead of a lambda, I define an instance method `:populate_user!` to be invoked for that (line 5).

As discussed in earlier chapters, the `:populator` option works similar to `:populate_if_empty` but is invoked at a lower level. Also, `:populator` is invoked in `validate`, whereas prepopulation gets triggered via `prepopulate!`

In a populator, you have to do everything by hand. It is always called, and you need to manually assign values to the form. This is a different to what we've learned with `:populate_if_empty`, which only gets invoked when the corresponding nested form is not found, and automatically assigns the result to the object graph.

In the `:populator` I try to find a `User` by the email that was submitted. Note that `fragment` represents the nested hash of the form. If the `find_by` is unsuccessful, I instantiate a new user. The instance must then be assigned via the respective setter, otherwise it will be lost and the nested user form will be empty (line 10).

In an instant, a test for this goes to `test/concepts/comment/operation_test.rb`. Otherwise, we're gonna forget we added this feature, bugs start coming, and the world is gonna explode. Or, at least, it stalls.

```

1  it do
2    params = {
3      id:      thing.id,
4      comment: {"body"=>"Fantastic!", "weight"=>"1",
5                 "user"=>{"email"=>"joe@trb.org"}}
6    }
7    op1 = Comment::Create.(params)
8    op2 = Comment::Create.(params)
9
10   op1.model.user.id.must_equal op2.model.user.id
11 end

```

An extremely simple test. I create two comments subsequentially with the same params (line 2-8). By asserting that the first and second operations' user model have the same ID I can make sure there was no additional user created in the second call (line 10).

Sleeping Users

Gemgem allows to create and update things and authors thereof. Authors, or users, won't be created twice. Once they exist in the database, the existing user will be associated to a thing if emails match.

A few checks are run to make sure users can only have a maximum of five unconfirmed authorships, and so on.

When commenting, the specified user will be created or an existing one found and associated analogue to how we did it with things authors. We won't have any duplicates of users in the database.

Whenever a user gets created “on-the-fly”, either when adding things or comments, the user will get notified to join us. Instead of calling this “unconfirmed” I named it “*sleeping user*”: An implicitly created user that exists in the system, has a thing or a comment associated to it, but whose account is not confirmed, yet.

I played with many names, “unconfirmed”, “needs password”, “deactivated”, just to name a few. “Sleeping user”, I have no idea why, makes me think of the correct state of affairs.

Instead of letting sleeping users confirm their account to become full-blown *Gemgemonites* I decided to walk you through the manual sign up process, first. The manual sign up allows users to explicitly register for Gemgem and then start playing with it.

We will explore the sign up screen and its validations, creating a user and signing in.

Tyrant

For the entire authentication system we refrain from using Devise. In chapter 2 I already discussed what makes me refuse this gem: It is lacking a clean object design and leverages global Rails concepts like callbacks, filters, and monkey-patching, techniques that are not appreciated in Trailblazer.

An alternative to Devise is [Tyrant](#)⁴⁸. This gem is an extraction from Gemgem and its foundation was laid when writing this book.

Tyrant comes with all common authentication features like sign in, sign up, password change, brute-force protection, and more. It does provide its functions as Trailblazer operations, its public API is exposed with twins, and views are implemented using Cells.

Validations, post-processing logic and internal behavior can be customized by using Trailblazer's polymorphism. In other words, you override operations, contracts and methods with plain Ruby instead of having to hack its source code.

Tyrant has zero coupling to Rails and can be used in any Ruby environment. Despite a high degree of encapsulation, it doesn't feel clumsy and fits smoothly into Rails applications.

To make you better understand how Tyrant works, we're going to implement two functions ourselves, and then I show you how this is done with Tyrant.

⁴⁸<https://github.com/apotonick/tyrant>



Sign Up

Let's discuss the sign up form and the processing first.

In Gemgem, all functions related to authentication will be invoked from the same controller `SessionsController`. In this controller, expressive actions will delegate to Tyrant operations. I gave up trying to fumble the different authentication functions into a “RESTful” style as I will discuss in a second.

As a first step, users will usually want to browse to a URL to hit the sign up form. I added a few manual routes to `config/routes.rb`.

```
1  get  "sessions/sign_up_form"
2  post "sessions/sign_up"
3  get  "sessions/sign_out"
4  get  "sessions/sign_in_form"
5  post "sessions/sign_in"
```

As you can see, I only use POST and GET routes where the names speak for themselves.

Breaking “RESTfulness”, part II

Again, I'm breaking Rails' “RESTfulness” and again I feel good about this. A controller action with the same named view `sign_up_form` is simply more intuitive than `create`. Likewise, a GET that can be sent from an HTML link to `sign_out` instead of a DELETE request to `destroy` makes more sense.

To the same effect, when adding more steps like “change password” or “activate account”, I am struggling to map those functions and workflows to resources with CRUD operations.

Authentication is a UI-specific workflow and has barely any overlapping with REST concepts. The opposite is the case: The central notion of our authentication is manifested in a cookie, an asset that has no place in the REST world at all.

Therefore, I create manual routes that are a tiny bit more explicit, I restrict myself to GET and POST requests and I actually understand what I do.

SignUp Operation

The sign up process will start with a form to enter your credentials. You've seen that kind of screen before. You enter your email, enter your desired password twice in two fields, click submit, and you're in.

That is, given you entered the password identically in both fields.

Here's the initial part of our first authentication operation. I put all operations related to this into `concepts/session/operation.rb`. Trailblazer will load this file automatically without trying to find and preload a corresponding model class.

This operation will represent the signup form, and the processing thereof.

```

1  class SignUp < Trailblazer::Operation
2    include Model
3    model User, :create
4
5    contract do
6      property :email
7      property :password, virtual: true
8      property :confirm_password, virtual: true
9
10   validates :email, :password, :confirm_password, presence: true
11   validate :password_ok?

```

This is a plain operation in good ol' Trailblazer style.

I include `Model` to let the operation create a `User` instance for process so we don't have to do that. Also, by specifying the model it will automatically work with `simple_form` as the operation and the form object now know their "model name" (line 2-3).

After specifying the `email` property, I declare both `password` and `confirm_password` as virtual (line 7-8). These fields don't exist on the `User` instance the form wraps and we only need them during the operation run. This is exactly what the `:virtual` option was made for.

All three fields can't be blank in the form submission, which is asserted with the `presence` validation (line 10).

The special `password_ok?` validation will compare `password` and the confirmed password. They need to be identical, but you already got that.

```

1  class SignUp < Trailblazer::Operation
2    # ...
3    def password_ok?
4      return unless email and password
5      return if password == confirm_password
6      errors.add(:password, "Passwords don't match")
7    end

```

Whenever the `SignUp` operation is run, the `password_ok?` will be run, too, regardless of whether or not earlier validations have failed. This is why I first check that `email` and `password` were actually submitted (line 4).

If the two submitted passwords are not identical, a message gets added to `errors`, marking the validation as invalid (line 5-6).

Creating a new User

After the validations have been sorted, I move on to processing. This means, in case of a successful form submission with valid input, we need to create a new user and assign the password to it, so they can login the next time.

In Gemgem, I decided to not use the confirmation workflow popular in many websites, where, after sign up, you need to click a confirmation link, first. This is handled by Tyrant and easy to integrate in your application, should you want that.

Gemgem lets you signup immediately. In trade, several new steps are to be implemented, for example, sleeping users need to be able to activate their account. I do all this non-conform behavior to show you how to customize Tyrant.

```

1 class SignUp < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:user]) do
5       create!
6       contract.save
7     end
8   end
9
10  def create!
11    auth = Tyrant::Authenticatable.new(contract.model)
12    auth.digest!(contract.password)
13    auth.confirmed!
14    auth.sync
15  end
16 end

```

The first part is the familiar invocation of the form validation. Note that the operation expects input under the `:user` key (line 4). In case of a successful validation, a mysterious method `create!` is called (line 5).

Where I could have used a callback, I decided a method before saving the contract will do.

The `create!` method uses a part of Tyrant's public API to accomplish its job of creating a user with a password. This public API is available via the `Tyrant::Authenticatable` twin. When instantiated, this class requires a user model, that's why I pass in `contract.model` (line 11).

Invoking `digest!` and passing in the submitted password will use the bcrypt algorithm to generate a password digest. This cryptic string is kept in the twin, nothing happens to the user instance, yet (line 12). I will discuss the internals of `Authenticatable` in a minute.

When calling `confirmed!` on that twin, all that happens is the twin will mark itself as confirmed - what a surprising thing, given the name of the method. That basically involves storing the confirmation date internally. Again, nothing happens to the user (line 13).

Only when calling `sync` on the `Authenticatable` twin, its internal state is written to the `User` instance. This will assign a hash containing all information necessary to the `User`'s `auth_meta_data` field, which is the only requirement Tyrant has to your user model.

To persist this state change and write the `auth_meta_data` hash to its field in the database, I call the contract's `save` (line 6). This will first sync the `email` field from the contract to the user instance since it is the only non-virtual property. It then calls `save` on the `User` instance which will persist the new email and the authentication data to a new row in the database.

This won't work, though, without defining `auth_meta_data` on the `User` class.

I created a migration to add this field.

```
1 class AddAuthMetaDataToUsers < ActiveRecord::Migration
2   def change
3     add_column :users, :auth_meta_data, :text
4   end
5 end
```

This will add `auth_meta_data` to the `users` table and allow arbitrary text in it (line 3).

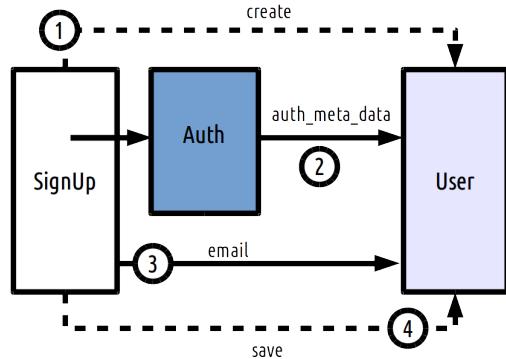
Since we will store a serialized hash in it, this needs to be configured on the model.

```
1 class User < ActiveRecord::Base
2   # ...
3   serialize :auth_meta_data
4 end
```

We already encountered the `serialize` class method in the last chapter. By declaring the `auth_meta_data` serializable, Rails will automatically render it to a JSON hash for persistence and parse it back to a hash when querying it. This works identical to the `image_meta_data` field used by Paperdragon.

Summary SignUp

To understand the overall control flow in `SignUp`, have a look at the diagram.



The operation and its form are responsible for creating the user object and validating the email (1).

Tyrant's **Authenticatable** twin is then called in the operation and computes the **auth_meta_data** field. The hash is written back to the model when the twin syncs (2).

Once the model is ready to be persisted, the operation takes back control and calls **save** on the user (4). Implicitly, the **email** field from the contract gets written to the user before it is persisted (3).

You might still be struggling to understand what exactly is the **auth_meta_data** field about. Why not have a look at the **Authenticatable** twin that comes with Tryrant?

Authenticatable

How does the twin generate and store the password for the user, and how does all the data get back to the database and associated to the user?

It's **Authenticatable**'s job to expose a small API of methods related to authentication. While this could have also been done with a set of operations, this all comes in one object. I use twins a lot with small groups of business logic - this is way easier to use than invoking clumsy operations for every step.

Here's, again, how the **Authenticatable** twin gets instantiated.

```
1 auth = Tyrant::Authenticatable.new(contract.model)
```

The **Authenticatable** twin wraps, or decorates, a **User** instance. This actually is more than just a decorator, as the twin also writes to the decorated object. We will see that in a minute.

Here's an excerpt from the **Authenticatable** class and its schema definition.

```

1 class Authenticatable < Disposable::Twin
2   property :auth_meta_data do
3     include Struct
4     property :confirmation_token
5     property :confirmed_at
6     property :confirmation_created_at
7     property :password_digest
8   end

```

As you can see, this is simply a subclass of `Twin`. It defines one property `:auth_meta_data` on the top level (line 2). This property is the only interface to the `User` model, `User#auth_meta_data` and `User#auth_meta_data=` have to be defined on the model.

In the nested twin, the `Struct` feature gets included making `auth_meta_data` a hash field (line 3). A handful of properties then let you divine what this all might be about (line 4-7).

Hash Fields

To understand `Struct` twins, or hash fields, we have to understand what happens when initializing the twin.

1. The `Authenticatable` twin will ask the decorated user instance for its `auth_meta_data` field. On the `User` model, this is a `serialize` field. Consequentially, when the twin invokes `user.auth_meta_data` this will return a hash, for instance `{password_digest: "abc"}`.
2. As we include `Struct` the twin knows this is going to be a hash, and it converts it into a decent nested twin object. This allows you to read the nested values via reader methods:

```
auth.auth_meta_data.password_digest #=> "abc"
```

3. Vice-versa, we can also write to all the defined properties using setter methods instead of clumsy hash fumbling.

```
auth.auth_meta_data.password_digest = "cba"
```

This won't write to the model, yet.

4. After working on the twin via its object-oriented API, we can write changes back to the model.

```
auth.sync #=> user.auth_meta_data = {password_digest: "cba"}
```

`Authenticatable` will compile `auth_meta_data` back to a hash and push it to the model by invoking the setter, as illustrated in the above snippet.

The twin provides a convenient API to the hash field and delays writing until you say so. By using a `Struct` twin I can be sure that there will only be one model write operation, once I call `sync`.

Authentication Logic on Twin

Besides abstracting the persistence layer, the twin is an OOP asset that allows to add behavior. In Tyrant, `Authenticatable` exposes a minimal API for handling authentication maintenance with a persisted object. Let's recall the `SignUp` operation's `create!` method once more.

```

1 def create!
2   auth = Tyrant::Authenticatable.new(contract.model)
3   auth.digest!(contract.password)
4   auth.confirmed!
5   auth.sync
6 end

```

You have an idea now what is actually happening here. For completeness, I want to run through the implementation of the methods used in this example.

```

1 class Authenticatable < Disposable::Twin
2   # ...
3   def digest!(password)
4     auth_meta_data.password_digest = BCrypt::Password.create(password)
5   end

```

The `digest!` method is the first method dispatched in `SignUp`. It digests the incoming form password using the bcrypt library and assigns it to a virtual field `password_digest` on the hash field.

Analogously, the `confirmed!` method doesn't do anything big, either.

```

1 class Authenticatable < Disposable::Twin
2   # ...
3   def confirmed!
4     auth_meta_data.confirmation_token = nil
5     auth_meta_data.confirmed_at      = Time.now
6   end
7 end

```

When invoking the `confirmed!` method in the `SignUp` operation, the twin will reset the `confirmation_token` field and mark it as confirmed by setting a `confirmed_at` timestamp (line 4-5).

The last step in `SignUp` is calling the `sync` method on the twin.

```
1 auth.sync
2 #=> user.auth_meta_data = {
3 #  confirmed_at: "2015-07-12",
4 #  confirmation_token: nil,
5 #  password_digest: "abc",
6 # }
```

The `Authenticatable` twin used in Tyrant is really simple and the implementation straight-forward. Nevertheless, it encapsulates any knowledge about the `auth_meta_data` field and only exposes methods to query or change state.

By using this twin, you don't need to think about the format or the meaning of the `auth_meta_field` at all, and this is a great improvement to Devise and its fellow gems.

Instead of letting users change arbitrary fields manually, probably altering state to an invalid combination, `Authenticatable` provides an API to reach any possible state without having to learn about the semantics of `auth_meta_data`, or, as in Devise, `password_digest`, `confirmation_token_sent_at` and all the other columns that are not covered by any API.

Now that we've learned everything, literally everything, of the internals of the `SignUp` process we should hook that into the controller and our UI. The next step is to render the `signup` form and process it.

SignUp Form

In the `SessionsController` action `sign_up_form` we implement the form where users enter email, password and password confirmation. This is for the file `app/controllers/sessions_controller.rb`.

```
1 class SessionsController < ApplicationController
2   def sign_up_form
3     form Session::SignUp
4   end
```

Presenting the sign up form couldn't be any simpler. We simply pass the `SignUp` operation to the controller's `form` method and move on to the view to render the fields.

The view resides in `app/views/sessions/sign_up_form.html.haml` and is a standard controller view.

```

1 %h1 Come join us!
2 = simple_form_for(@form, url: sessions_sign_up_path) do |f|
3   = f.input :email
4   = f.input :password
5   = f.input :confirm_password
6   = f.button :submit, "Sign up!"

```

Again, this is so simple I won't waste any paper. As always, the form object of the operation is available as `@form`. Also, I need to point `simple_form` to the correct processing `:url` since I do not use "RESTful" style resources that do not make any sense here (line 2).

When filled out and submitted, the form gets validated and processed in `SessionsController#sign_up`.

```

1 class SessionsController < ApplicationController
2   # ...
3   def sign_up
4     run Session::SignUp do |op|
5       flash[:notice] = "Please log in now!"
6       return redirect_to sessions_sign_in_form_path
7     end
8
9     render action: :sign_up_form
10    end

```

This time, I run the `SignUp` operation (line 4). When valid, a physical `User` gets created as discussed earlier, and the user gets redirected to the login page where they will see a message telling them to sign in (line 5-6).

If invalid, the form gets rerendered using the `sign_up_form` view and displays errors, e.g. when the passwords mismatch (line 9).

Testing SignUp

I have written many tests for signup since I was nervous. What would the suits tell me if users could login without being authorized? Better to write a few tests too many. That's why I test the `SignUp` operation in `test/concepts/session/sign_up_test.rb`.

```

1 class SignUpTest < MiniTest::Spec
2   # valid signup.
3   it do
4     res, op = Session::SignUp.run(user: {
5       email: "selectport@trb.org",
6       password: "123123",
7       confirm_password: "123123",
8     })
9
10    op.model.persisted?.must_equal true
11    op.model.email.must_equal "selectport@trb.org"
12    assert Tyrant::Authenticatable.new(op.model).digest == "123123"
13  end

```

Running the operation with valid input, I assume that the user model got persisted (line 4-10). I also make sure the email was set correctly (line 11). To make it 100% safe, I pass the created user model to `Authenticatable` and compare `digest` with the real password (line 12).

Internally, the bcrypt library will now encrypt "123123" the same way it encrypted the old password, and compare it to the `auth_meta_data.password_digest` field. How that works, I have no idea, but again, I see the benefit of the `Authenticatable` twin and how it hides hideous details from me.

In the test file, I have several more invalid cases tested which really do not need to be discussed here as they're all straight-forward.

Sign In

The manual sign up works. Users can enter their credentials and when valid they get redirected to sign in screen. That we need to implement now.

As a reminder, here are the two routes that connect our new `SignIn` operation to a form and processing action.

```

1 get  "sessions/sign_in_form"
2 post "sessions/sign_in"

```

Looking at the `sign_in_form` action of the `SessionsController` you will see that this is really just another form-rendering action.

```

1 class SessionsController < ApplicationController
2   # ...
3   def sign_in_form
4     form Session::SignIn
5   end

```

For completeness, I want to run through the contract definitions and the controller view to render the sign in form, real quick.

Modelless Forms

The `Session::SignIn` operation I put into the `app/concepts/session/operation.rb` file where we also keep `SignUp` and friends.

```

1 module Session
2   class SignIn < Trailblazer::Operation
3     contract do
4       property :email,    virtual: true
5       property :password, virtual: true

```

The only fields declared in the operation's contract are `email` and `password` (line 4-5). Both are virtual, meaning they won't be read from the model (and will not be written back in `#sync`).

In our case, this means the contract can be initialized with a `nil` model and will still work as no values are attempted to be read from the "model". This in turn implies our contract can successfully be rendered in the login screen even though we don't have a model for the contract.

When initializing the contract in the controller using the `form` method, what basically happens is the following.

```

1 def form(operation_class)
2   @form = operation_class.contract.new(nil)
3 end

```

This is, of course, not the real implementation but shows all necessary steps. As we didn't define any model for this operation, the contract will be instantiated with `nil`. This works since all its fields are virtual.

The corresponding controller view `app/views/sessions/sign_in_form.html.haml` is incredibly sophisticated.

```

1 = simple_form_for(@form, url: sessions_sign_in_path, as: :session) do |f|
2   f.input :email
3   f.input :password
4   f.button :submit

```

Most of the work goes into urging SimpleForm not to wildly assume names and routes, which I find quite inconvenient and makes me want to write my own form builder on some days. I configure the `:url` option and also need to tell the form builder to put all form fields under the `:session` key using the `:as` directive (line 1).

I allow the user to enter email and password, and a submit button makes sense, too, here (line 2-4). Rendering forms is boring. Let's see how we handle the login process in our `SignUp` operation, the controller and Tyrant.

Login

When submitting, the form gets sent to `SessionsController#sign_in`. Again, this is a simple delegation to the sign in operation.

```

1 class SessionsController < ApplicationController
2   def sign_in
3     run Session::SignIn do |op|
4       tyrant.sign_in!(op.model)
5       return redirect_to root_path
6     end
7
8     render action: :sign_in_form
9   end

```

In case of an invalid operation run, the user will be presented with the login form, again (line 8). If the operation is valid, a block of code is executed. First, a strange `tyrant` object is used and apparently executes the login process, and then the user gets redirected to the home page (line 4-5).

Before discussing this *logic* in the controller in more detail, we should explore how the operation finds out about its validity. Here's the rest of the `SignIn` contract.

```

1 class SignIn < Trailblazer::Operation
2   contract do
3     property :email,    virtual: true
4     property :password, virtual: true
5     validates :email, :password, presence: true
6     validate :password_ok?
7
8     attr_reader :user
9   private
10  def password_ok?
11    return if email.blank? or password.blank?
12    @user = User.find_by(email: email)
13    errors.add(:password, "Wrong password.") unless @user and \
14      Tyrant::Authenticatable.new(@user).digest?(password)
15  end
16 end

```

In addition to the properties we've already met there is a presence validation for both email and password to make sure these are always filled out (line 5). A custom validation named `password_ok?` will do the main work of this form.

First, I need to verify that both fields are filled out, again (line 11)⁴⁹.

Then, something strange: I use the `User` model's finder to actually retrieve an object from the database. Since we need the model for verifying the password, there currently is no other place than the validation to find this (line 12).

While this could happen in the operation, too, you'd have to access `params` there by hand to get the email. I don't like doing this as I prefer relying on the form's deserializing and then conveniently grab the submitted email via the contract's `email` reader.

Also note that I assign the user object to an instance variable and expose a public reader for that (line 12 and 8).

The last line is the actual validation. I check that a user was found and hand it to `Authenticatable` where I use the `digest?` method and pass in the submitted password (line 13-14). If `Authenticatable` decides this is the correct password the validation will be valid, otherwise, the contract and operation are marked as invalid.

Form and validations are completed, the last step is the `SignUp#process` method.

⁴⁹This is very likely to get replaced with nested validations that will soon be available in Reform and make chained conditional validations simpler.

```

1 class SignIn < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:session]) do |contract|
5       @model = contract.user
6     end
7   end
8 end

```

The validations we just coded are run using the operation's `validate` method (line 4). I pass in the `:session` key of the `params` hash because I named the form `session`, as explained earlier.

If the validations are successful and valid, all I do is assign the operation's instance variable `@model` by copying the user from the contract (line 5). This is simply the `User` instance we used in the validation.

We processed the form submission, validated the data, found a user object, all we need now is to log it in. This happens in the controller.

```

1 def sign_in
2   run Session::SignIn do |op|
3     tyrant.sign_in!(op.model)
4     return redirect_to root_path
5   end

```

Remember, the block passed to `run` is only invoked for a successful operation with a valid state. I simply pass the model, originally from the operation's contract, to the `sign_in!` method (line 3).

Now, where does this `tyrant` object come from? You will agree with me that every application needs a `tyrant`, but what exactly is that?

Application-wide Tyrant

Signing in a user works by passing a cookie to the browser after a successful login. The cookie will then, per request, authenticate the browser with a certain user in the backend.

Cookies are very HTTP-specific concepts and I believe that operations should not know about this low-level mechanism unless you really need to replicate this behavior in other environments of your application, for instance, in a cookie-backed document API - which is not really what REST is about.

That being said, I made three technical decisions for Gemgem authentication.

1. The app-wide `tyrant` object acts as binding between application and authentication. It is managed in `ApplicationController` as controllers are the place where we may deal with the transport layer (HTTP) to access cookies.

2. Physically invoking sign in and sign out, too, happens in the controller. In the `sign_in` action, you can see how the `tyrant` object is used to log in the user. While this could go into operations, I like it better here.
3. Sign-in specific authorization, e.g. showing or rejecting the sign in form, goes into a `before_filter` in the involved controller. As we will learn in the next chapter, a policy could be used here, too. However, this is so HTTP-specific that I won't bother my operation code with it.

```

1 class ApplicationController < ActionController::Base
2   # ...
3   def tyrant
4     Tyrant::Session.new(request.env['warden'])
5   end
6   helper_method :tyrant

```

The `tyrant` method is the first “helper” I add to Gemgem. It returns a `Tyrant::Session` object and as input it requires a part of the request (line 4). The `request` is only available in controllers which is why I allow this single helper to be here.

As our views are not completely cells-based, this method is also allowed to be called in views (line 6).

The `tyrant` object exposes a minimal, super simple API to manage authentication. Its public methods are `sign_in!`, `sign_out!`, `signed_in?` and `current_user`. Behind the scenes, `Session` will implement these functions using the global Warden object.

Recall the `sign_in` action of our sessions controller.

```

1 run Session::SignIn do |op|
2   tyrant.sign_in!(op.model)
3 end

```

Not sure if that needs any explanation. If the `SignIn` operation decided that the email/password tuple is valid, the user found in the operation is signed in using `sign_in!` (line 2). From now on, a cookie handled by Warden will sit in the user's browser and authenticate the session in each request, providing us with a `current_user` object.

Before_filter

Sigining in should only be allowed for sessions that are not logged in, yet. This isn't really security-relevant but a good time to discuss why I very rarely use a `before_filter` to protect entire controller actions from unauthenticated access.

```

1 class SessionsController < ApplicationController
2   before_filter only: [:sign_in_form, :sign_in] do
3     redirect_to root_path if tyrant.signed_in?
4   end

```

The `tyrant` object is queried if a user is signed in. If this is true, requests to `sign_in_form` and `sign_in` are redirected to the home page and prevent “double logins” that theoretically wouldn’t hurt at all, but anyway (line 2-4).

Why am I using a filter here? Didn’t we say “*No business logic in controllers!*”?

Think of an operation as a reusable function. Reusable not because you could use them across different applications, but reusable within your Rails app in different environments. You will surely remember how we use operations in controllers and in tests as factories and how great this feels.

All operations in the `Session` namespace are relevant to HTTP environments, though. It really doesn’t make sense to call them on the command line as there is no concept of authentication in CLIs. Likewise, this doesn’t apply to unit tests.

Don’t confuse *authorization* with *authentication* here! We will learn at the end of this chapter and in the following one how we definitely need to differentiate between different user roles. Nevertheless, this has nothing to do with HTTP cookies.

In other words: Technically, I could have implemented the `SignIn` operation in the controller itself as it won’t be used anywhere else. I would have to replicate the form behavior, though, and hence use an operation here too, but leave HTTP-specific logic in the controller.

This is why filter and the actual sign in code sits in the controller.

A Warm Greeting

Now that we can login, it’ll be cool to have some kind of visible indicator that we’re signed in. I add a welcome message to the navigation bar in `views/layouts/_navigation_links.html.haml`.

```

1 %li
2   = link_to "Start discussion!", new_thing_path
3 - if tyrant.signed_in?
4   %li
5     = link_to "Hi, #{tyrant.current_user.email}", user_path(tyrant.current_user)
6   %li
7     = link_to "Sign out", sessions_sign_out_path
8 - else
9   %li
10    = link_to "Sign in", sessions_sign_in_form_path
11   %li
12    = link_to "Sign up", sessions_sign_up_form_path

```

A classic Rails view here does the trick. With a rather ugly if/else block I put two different contexts into the same view. Admittedly, this is becoming unpretty and should be refactored to a cell. Soon! I use the `tyrant` helper that we defined earlier in `ApplicationController` and differentiate between signed in and public user (line 3 and 9). The “`Hi, jonny@trb.org`” string is only rendered for a signed in user (line 5).

This is a massive change to the view and requires a functional test. We will write it shortly when we test sign in and out.

Signing Out

Before we see this all in action in one of our wonderful integration tests, here’s a quick rundown of signing out a user. This is so simple I won’t waste more than half a page for this - let’s save another tree, instead.

```
1 get "sessions/sign_out"
```

I add one route to the `sign_out` action of the sessions controller. Note that this is done with a simple GET so we can easily sign out following a link. If you still think you have to implement that with a fake DELETE request to a session “resource”, feel free to hook `SignOut` into any action you like.

```
1 class SessionsController < ApplicationController
2   # ...
3   def sign_out
4     run Session::SignOut do
5       tyrant.sign_out!
6       redirect_to root_path
7     end
8   end
```

Given the `SignOut` operation is valid, which is currently always true, we sign out the session via the `tyrant` object (line 3).

The operation itself is empty.

```
1 class SignOut < Trailblazer::Operation
2   def process(params)
3   end
4 end
```

No validations or processing are found here. I still map this to an operation in case we want to add behavior, for example, logging the sign out of the user. Mainly, this operation is here for consistency.

Testing Logins

As I won't use SignIn other than for the controller, I test the entire sign in/out process via an integration test, only. This will give me maximum security and since there's not many edge cases, a few smoke tests won't be too hard to implement.

Both sign in and sign out go into `test/integration/session_test.rb`. Here, I will only discuss the happy path, but you can find several negative tests in the repository.

```
1 class SessionIntegrationTest < IntegrationTest
2   # ...
3   it do
4     visit "sessions/sign_up_form"
5     submit_sign_up!("fred@trb.org", "123", "123")
6     submit!("fred@trb.org", "123")
7
8     page.must_have_content "Hi, fred@trb.org" # login success.
9
10    # no sign_in screen for logged in.
11    visit "/sessions/sign_in_form"
12    page.must_have_content "Welcome to Gemgem!"
```

I first sign up and log in using the UI (line 4-6). In order to do so, I wrote two helper methods `submit_sign_up!` and `submit!` to automate form submissions. These helpers sit directly in the test class and you can find them in the repository.

Sending the correct credentials I assert that the page now shows the welcome message in the navigation bar (line 8). This is enough test code to make sure logging in works. I explicitly do not check for cookies or anything hidden but only test for strings or selectors that I could "see" in a real click test.

After that, I make sure the sign in screen isn't accessable for signed in users and we get redirected to the home page (line 11-12)

```
1   click "Sign out"
2   page.current_path.must_equal "/"
3   page.wont_have_content "Hi, fred@trb.org"
4 end
```

As a last step, I click the sign out link which should redirect me back to the home page (line 1-2). Again, I don't check for hidden values but simply make sure there ain't no greeting on the page, anymore (line 3).

Putting Users to Sleep

In the last part of the chapter I want to focus on sleeping users and, of course, waking them up and activating their account.

When we implicitly create users when commenting or adding authors to things, so far we literally only “create” them without marking them as sleeping. This needs to be fixed before we move on to waking them up.

In the app/concepts/comment/operation.rb file, I add a callback to put the user to sleep once it got created when commenting.

```

1 class Comment < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     callback do
4       on_change :sign_up_sleeping!, property: :user
5     end

```

Using a callback block I instruct the operation to invoke `sign_up_sleeping!` when a change was detected (line 4). However, this “change” is limited to the `user` property of the contract twin by using the `:property` option.

The `:property` option in combination with `on_change` is a nice trick to trigger a callback only when a nested property has changed. In our example, this callback is always invoked since we always have to add an email to the comment’s user, which will be regarded as “changed”.

The implementation of the callback goes into the `Comment::Create` operation.

```

1 def sign_up_sleeping!(comment, *)
2   Tyrant::Authenticatable.new(comment.user.model)
3   auth.confirmable!
4   auth.sync
5 end

```

Since the callback method always receives the form twin it was detected on, I invoke `comment.user.model` to retrieve the actual user model and pass it to `Authenticatable` (line 2). The `confirmable!` method will take care of marking this model as “sleeping” (line 3). In order to write that to the model, I call `sync` on the twin (line 4).

The user model’s `auth_meta_data` hash now contains all data needed to interpret this user as confirmable, or “sleeping”, as we call it.

As a last step, we need to actually trigger the callback group so the `on_change` gets detected and the above logic gets run. As always, this happens in the operation’s `process` method.

```

1 def process(params)
2   validate(params[:comment]) do |f|
3     dispatch!
4     f.save # save comment and user.
5   end
6 end

```

The call to `dispatch!` will invoke the default callback group we just created, find the user object that has changed and call `sign_up_sleeping!` (line 3).

Callback in Thing::Create

The analogue we need to implement when adding authors to things. Again, I add a callback to `Thing::Create` in its callback file `app/concepts/thing/callback.rb`.

```

1 module Thing::Callback
2   class BeforeSave < Disposable::Callback::Group
3     on_change :upload_image!, property: :file
4
5     collection :users do
6       on_add :sign_up_sleeping!
7     end

```

In addition to the `:upload_image!` callback that was there before, I add `sign_up_sleeping!` (line 5-7). Here, I use the `on_add` hook in the `users` collection which will trigger this callback for every user that has been added in `process`.

The callback itself is very similar to the one in `Comment::Create`.

```

1 class BeforeSave < Disposable::Callback::Group
2   # ...
3   def sign_up_sleeping!(user, *)
4     return if user.persisted? # only new
5     Tyrant::Authenticatable.new(user.model)
6     auth.confirmable!
7     auth.sync
8   end

```

First, I need to check whether or not this user has been created just now. I do this by calling `persisted?` on the user twin, which will return false if the model hasn't been written to the database, yet (line 4). That in turn means the user is new and needs to sleep, as it has been created implicitly.

The rest of the code is identical to the callback above. In the next chapter, we will extract common logic to a separate class.

Since the `:before_save` callback already gets invoked in `Thing::Create#process`, no other changes are needed here.

For both cases, commenting and adding authors to things, new users will now be marked as sleeping, or, as Tyrant calls it, “confirmable” and will allow them to set a password and activate their account.

Waking Up Sleeping Users

We covered all major concepts of authentication: signing up new users, implicitly creating “sleeping users” when commenting and when adding new things, signing in and out, the only function missing to finish this chapter is allowing users to activate their sleeping account.

This works by following the link in the email that points to `SessionsController#wake_up` and passes on the confirmation token. If the latter is correct and matches the sleeping user’s token, a form is displayed to let the user set a password. Submitting two identical passwords will activate, or “wake up” the user’s account.

Here’s the two routes I added to implement the form and the processing endpoints.

```
1 get "sessions/wake_up_form/:id"
2 post "sessions/wake_up/:id"
```

I left out some noise required by Rails’ routing weirdness. The wake up link will point to the `wake_up_form` controller action and have a format similar to the following.

```
1 /sessions/wake_up_form/1/?confirmation_token=abcabc
```

Both user id and confirmation token are embedded in the URL. The action first needs to check whether the confirmation token is valid and only then render the form to set the password.

```
1 class SessionsController < ApplicationController
2   # ...
3   before_filter only: [:wake_up_form] do
4     Session::IsConfirmable.reject(params) { redirect_to(root_path) }
5   end
6
7   def wake_up_form
8     form Session::WakeUp
9   end
```

With a `before_filter` I restrict access: only when params contains the correct `confirmation_token` for the provided user, the form is rendered. Otherwise, a redirect to the home page happens (line 3-5).

Validating Operation

I use an Operation here to validate the confirmation token. This is used in other places, too, so it is legit to model that behavior in an operation. Note that I use Operation's `reject` class method here instead of `run`. `reject` will do the exact same thing as `run` but execute the passed block when the result is invalid.

In the actual controller action, we use `form` to render the password form (line 7-9).

Let's see what the `IsConfirmable` operation looks like and then inspect the password form.

```
1 module Session
2   class IsConfirmable < Trailblazer::Operation
3     include CRUD
4     model User, :find
5
6     def process(params)
7       return if Tyrant::Authenticatable.new(model).
8         confirmable?(params[:confirmation_token])
9       invalid!
10    end
11  end
```

Since the incoming params hash contains the sleeping user's ID I use `CRUD` to do the model finding for me (line 3). This won't work unless I configure that this is a find-only operation (line 4).

When running the operation, the user model will automatically be available via `model` (line 7). I use `Authenticatable` to find out whether or not this user is confirmable. The `confirmable?` method requires the token to be passed in and will then check the `auth_meta_data` field for the `confirmation_token`, do the comparison and return the result (line 8).

If this is all correct, I return (line 7-8). This will leave the operation as valid and thus won't trigger a redirect in the `before_filter`.

Otherwise, I invoke `invalid!` to mark the operation as failed. This will redirect the browser prohibiting access to the change-password form.

WakeUp

To understand the rendering of the form we need to have a look at the `WakeUp` operation, first.

```

1 module Session
2   class WakeUp < Trailblazer::Operation
3     include CRUD
4     model User, :find
5
6     contract do
7       property :password, virtual: true
8       property :confirm_password, virtual: true

```

Again, this uses CRUD and finds the user object according to the :id field in the incoming params (line 3-4).

The contract then defines two fields password and confirm_password, both virtual, to allow the user to enter their desired passphrase (line 7-8).

The form view in app/views/sessions/wake_up_form.html.haml is simple, too.

```

1 = simple_form_for(@form, url: session_wake_up_path(id: params[:id])) do |f|
2   = f.input :password
3   = f.input :confirm_password
4   = f.button :submit, "Engage!"
5   = f.input :confirmation_token, as: :hidden,
6     input_html: {value: @operation.confirmation_token}

```

The form's URL points to SessionsController#wake_up to initialize the user's password. Since this URL needs a user ID, I directly access params[:id] which is a bit dirty but does the trick for now (line 1). Usually, I never allow views to access params and I really should do this via the operation or the form.

I add the two password fields and a submit button (line 2-4). What's more important is that I add a hidden field named confirmation_token (line 5-6). Its value I grab from @operation.confirmation_token (line 6) and we will see how that is set in a minute.

The reason I do this is to allow users to resend the form should it be invalid. The wake_up action always needs a confirmation token to be sent, so we have to embed it into the form using this trick.

Processing Wake Up

When submitted, the wake_up controller action is hit.

```

1 class SessionsController < ApplicationController
2   def wake_up
3     run Session::WakeUp do
4       flash[:notice] = "Password changed."
5       redirect_to sessions_sign_in_form_path and return
6     end
7
8     render :wake_up_form
9   end

```

If the `WakeUp` operation wasn't run successfully, the form is rerendered (line 8). Otherwise, a valid processing will redirect the awaken user to the login page (line 5). I added a flash message to indicate my excitement about this very moment (line 4).

Let's now discuss the rest of the `WakeUp` operation.

```

1 class WakeUp < Trailblazer::Operation
2   # ...
3   contract do
4     property :password, virtual: true
5     property :confirm_password, virtual: true
6     validates :password, :confirm_password, presence: true
7     validate :password_ok?
8
9     def password_ok?
10    return unless password and confirm_password
11    errors.add(:password, "Password mismatch") if password != confirm_password
12  end
13 end

```

A presence validation makes sure both fields are filled out (line 6). The custom validator `password_ok?` will check if both passwords match (line 7 and 9-13). I will replace this redundant code with a module later, as this is almost identical and copied from `SignUp`.

The next part of the `WakeUp` operation will validate and set passwords in case of success.

```

1 class WakeUp < Trailblazer::Operation
2   # ...
3   def process(params)
4     validate(params[:user]) do
5       wake_up!
6     end
7   end
8
9   def wake_up!
10  auth = Tyrant::Authenticatable.new(contract.model)
11  auth.digest!(contract.password)
12  auth.confirmed!
13  auth.sync
14  contract.save
15 end

```

A private method `wake_up!` only gets invoked when validations returned a true result (line 4).

To set the password and confirm the user, I use `Authenticatable` and pass in the `User` instance of the operation (line 10). Using our old friend `digest!` the password is set (line 11). `confirmed!` marks the user as “awake” (line 12). A `sync` on the `Authenticatable` instance writes the `auth_meta_data` to the user model and `save` on the contract will persist all changes made to the user (line 13-14).

In order to allow the form view to render the confirmation token, we need to add a bit of code here. This time, I won’t access `params` but provide it from the operation. I do this to demonstrate different ways of how to access data from the operation.

```

1 class WakeUp < Trailblazer::Operation
2   # ...
3   attr_reader :confirmation_token
4   def setup_params!(params)
5     @confirmation_token = params[:confirmation_token]
6   end

```

In the `setup_params!` hook I simply grab the respective `params` element and assign it to an instance variable of the operation object (line 5). By exposing a public reader, this can be read in the view (line 4).

As long as the `params` value and the operation’s token don’t diverge, you can safely use the `params` hash directly in the view. However, often, data gets further processed in an operation and then you shouldn’t use global data but access it via the operation. Or, and that’s the clean way, with a cell, as we will learn it in the next chapter.

Integration Tests for Wake Up

The wake up function deserves tests. Again, all testing happens on the top-level in integration tests as we don't use any of the functions outside the HTTP environment, yet.

This will change in the next chapter, by the way, where we will add some admin pages and expose functionality like “*Wake up user!*” to the web UI.

Tests for the wake up operation go into `tests/controller/session_integration_test.rb`. I only explain one test case, the happy path. The failing ones are in the repository and so simple that we don't discuss them here.

```

1 class SessionIntegrationTest < IntegrationTest
2   it do
3     user = Thing::Create.(thing: {
4       name: "Taz", users: [{"email"=> "fred@taz.de"}]
5     }).model.users[0]
6
7     token = Tyrant::Authenticatable.new(user).confirmation_token
8
9     visit "sessions/wake_up_form/#{user.id}?confirmation_token=#{token}"
10
11    page.must_have_content "account, fred@taz.de!"
12    page.must_have_css "#user_password"
13    page.must_have_css "#user_confirm_password"
```

In the first part of the test I create a thing with an implicit user using the `Thing::Create` operation (line 3-5). Note how I grab the sleeping user using the `users` association on the thing model.

`Authenticatable` then helps me to grab the confirmation token from the newly created user (line 7).

I then compute the wake up URL by hand and open this page (line 9). Admittedly, I could utilize a URL helper here, but since I won't change the URL layout, I find the compilation doesn't break any DRY principles.

In the following three tests I make sure that we really see a positive “*Activate your account, [email]!*” header and the form (line 11-13).

```
1 fill_in "Password", with: "123"
2 fill_in "Password, again", with: "123"
3 click_button("Engage")
4
5 page.must_have_content "Password changed."
6 Tyrant::Authenticatable.new(user).confirmed?.must_equal true
7
8 page.current_path.must_equal "/sessions/sign_in_form"
```

By entering two identical passwords and submitting the form, I hopefully wake up my sleeping author (line 1-3).

I then check the flash message (line 5). Again, `Authenticatable` comes to help me with checking if the user is confirmed, now, which should be the case after submitting the password form successfully (line 6).

By testing the current path I assert that we're now on the sign in page and ready to log in (line 8).

```
1 fill_in "Email", with: "fred@taz.de"
2 fill_in "Password", with: "123"
3 click_button "Sign in!"
4
5 page.must_have_content "Hi, fred@taz.de"
```

The last part signs in using the form (line 1-3). It is followed by the same tedious test we had a hundred times before where I make sure the welcome message is visible (line 5).

Hang on, did we really finish this chapter? It seems so, because we now create sleeping users implicitly in several functions in Gemgem. These users can then, later, confirm their account.

They can also sign up manually, and we have a beautiful login page in place. And, eventually, when people get sick of Gemgem, there's a sign out button.

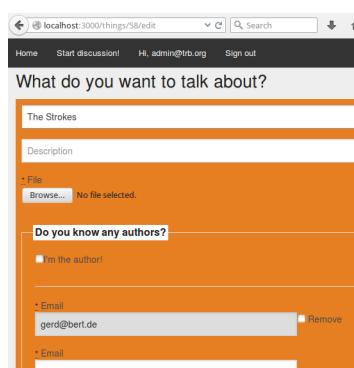
The amount of code needed for this is surprisingly little. This is partly because we reuse a lot of Trailblazer's structuring and can focus on the authentication logic instead of figuring out where to put this or that code.

Another reason is Tyrant. We do not use many features of Tyrant, yet, but it already saved us a lot of hassle. All logic related to authentication management, like password, confirmation flags, and tokens are encapsulated by `Authenticatable`. And in the next chapter, we will learn how to save even more code by simply subclassing Tyrant operations and customizing the well tested classes from this gem.

Speaking of the next chapter! Let's move on and learn some more things about authorization in business logic, views and operations.

Authorization And Polymorphism

Now that we can log in and out, sign up as new users or wake up sleeping users that were implicitly created, it is about time to add some more user-specific features. This chapter discusses how we can use polymorphic techniques to change the look and behavior of application functions for specific users.



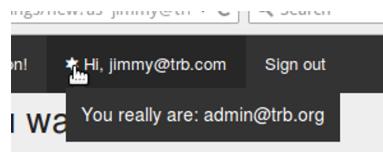
Signed-in users will see a “*I’m the author!*” checkbox when creating or editing a thing, making them an author without having to fill in their email address into one of the author fields.

Consequently, signed-in thing authors, and only those, can edit and delete things. They will see links to do so on the thing show page, but again, only for things they authored.

Anonymous users won’t be able to update or delete things anymore, the only allowed action is create.

We then introduce the concept of an admin user. The admin can edit and delete any thing. As a superuser, it can even change the name of a thing after creation, a feature that we restrict to admins exclusively.

The forms will have a conspicuous orange background when logged in as admin to remind you of your power, acting as a nice example how not only operation logic can be changed according to the configured rules, but also views.



A cool feature that I found extremely helpful while writing Gemgem is “impersonation”. As an admin user, you will be able to live out your multiple personalities to the fullest, as you can switch users on the fly while browsing. The application will now treat you as the impersonated user with the respective subset of

rights.

The navigation menu will display a star with a tooltip showing who you actually are and who you’re pretending to be. Crazy stuff, I know.

Policies

The first step when introducing user-specific functionality in a Trailblazer app is usually adding a policy to every operation that needs access control. A policy is a Ruby class that contains methods which represent rules. Policies have access to the current user and an arbitrary model and can thus compute permissions.

The way policies are designed in Trailblazer was greatly inspired by the excellent [Pundit⁵⁰](#) gem that introduced the idea of policy classes in Ruby. However, in Pundit policies are usually installed per model, whereas in Trailblazer, you have policy classes per concept or per operation.

Here's how the `Thing::Create` operation looks after I add a policy.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     include Policy
4     policy Thing::Policy, :create?

```

The `Policy` module comes from Trailblazer and needs to be included to import the `policy` class method and additional logic that is discussed in the following section (line 3).

To activate the policy, I reference the policy class `Thing::Policy` and the rule that has to pass (line 4). Before we start learning how that all plays together, let's have a quick look into the policy class which is located in the concept directory at `app/concepts/thing/policy.rb`.

```

1 class Thing::Policy
2   attr_reader :model, :user
3
4   def initialize(user, model)
5     @user, @model = user, model
6   end
7
8   def create?
9     true
10  end
11 end

```

A policy object does nothing fancy. It is to be initialized with the user and model (line 4-6), you can query different rule methods on it and expect a boolean return value representing whether or not to allow this action in the context of the user/model tuple.

Our first rule `create?` will always return `true`, which is why I don't even access `user` or `model` in the rule method. That means any user, even not signed-in, can run the `Create` operation.

Creating and evaluating the policy object is all done automatically by the operation. You are wondering now, where is that policy object instantiated, how do user and model get into it, where is the policy applied and what happens when it returns `false`?

Imagine we were running the `Thing::Create` operation with the following input.

⁵⁰<https://github.com/elabs/pundit>

```
1 Thing::Create.(current_user: @user, thing: {name: "Rails"})
```

Without a policy, the operation's `setup!` method would create a fresh model, the subsequent process method would run your business code and we're done.

However, with a policy being configured, things work a tiny bit different.

The important part for us right now is: the policy is run in the operation's `setup!` method which is called right after the operation object got instantiated, but before process is invoked.

Here's roughly what it looks like when running the operation. Please note that this code just illustrates the workflow.

```
1 class Trailblazer::Operation
2   def setup!(params)
3     policy = Thing::Policy.new(params[:current_user], model)
4     policy.create? or raise NotAuthorizedError
5     #
6   end
```

As you can see, both the current user and the model are passed into the policy constructor (line 3). Per default, the operation will use the `:current_user` hash key from `params` to find the signed in user. This means, in case you run an operation manually, you need to pass in this object using the correct hash key.

After the policy creation, the configured rule method is called (line 4). If the result is falsey, an exception is raised.

The fact that the policy is evaluated in `setup!`, potentially throwing a `NotAuthorizedError`, will prevent your business code in `process` being run in case of a policy violation.

Coming back to our example, this particular policy will always pass since the `create?` method simply returns true. This means, the operation can be run regardless whether or not anyone is signed in.

Operation Inheritance

Even though this policy is more than boring and nothing exciting is gonna happen, I add it on purpose to `Create`. First, it is good to have some convention and I find it easier to add an empty policy which can then later be changed than having to rewire things once you need restrictions.

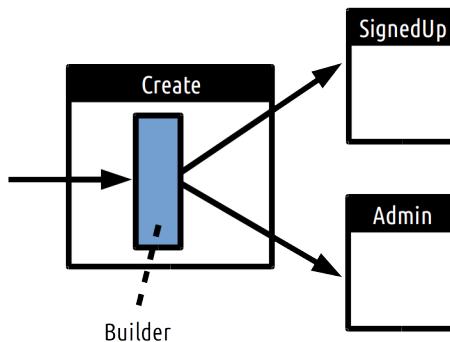
Second, policy objects in Trailblazer are not only useful for restricting access per operation. They are also extremely helpful in *builders* that resolve which concrete operation subclass to instantiate.

Hang on, what subclasses am I talking about? So far, we only inherit between different function operations. The `Thing::Update` class is derived from the `Create` class and inherits methods, contracts and the other components that an operation orchestrates.

Context

Builders, whatsoever, are meant to differentiate between different contexts for one function and then compute the matching subclass.

A *context* is usually referring to different user types like admins or moderators. With a different logged in user *context* comes a different behavior of UI and application. Nevertheless, contexts can also be the refinements for differing model types in case you're using STI, where subclasses change their semantics.



In this chapter, we are going to exploit the polymorphic characteristics of Trailblazer and take advantage of how subclassing or composites allow to refine a generic operation to a concrete context.

For instance, `Create` will have two subclasses `Create::SignedIn` and `Create::Admin` to fine-tune the contract and callbacks to the respective user type.

Allow me to quickly add these two new subclasses in order to demonstrate you how polymorphic operations and builders simplify your code.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     # ...
4     class SignedIn < self
5     end
6
7     class Admin < SignedIn
8   end
9 end
  
```

Here, we have the good ol' `Create` operation in all its beauty. The `SignedIn` class I put inline into the `create` class. That way, I will get the desired name `Thing::Create::SignedIn`. By inheriting from `self` it will be derived from the containing class (line 4). I could have simply said `class SignedIn < Create` but using `self` is less redundant and part of my OCD.

Likewise, `Admin` inherits from `SignedIn` and is namespaced as `Thing::Create::Admin`.

As we learned in chapter 3, inheriting from another operation will copy all methods to the new class in a plain Ruby way. In addition to that, the contract, callbacks, policies and representers will be copied and can be refined in the subclass without affecting the original.

Polymorphic Builders

The point about using subclasses and polymorphism is the avoid ugly `if/else` deciders throughout your code as seen in every Rails application. We will soon see how much better organized our code is by applying simple OOP instead of maintaining all contexts in one class and then manually dispatch and modify semantics with deciders.

The caller of your polymorphic operation doesn't even know that there's different subclasses to handle different contexts. *Builders*, a concept that originates from the Cells gem and proved to be very popular, takes the configuration of what subclass to create into the super class.

Applying this new concept to `Thing::Create` will change the class header a bit.

```

1 class Create < Trailblazer::Operation
2   include Resolver
3   policy Thing::Policy, :create?
4   model Thing, :create
5
6   builds -> (model, policy, params) do
7     return self::Admin if policy.admin?
8     return self::SignedIn if policy.signed_in?
9   end

```

Inside the `builds` block is where different contexts get identified and then dispatched to the respective subclass (line 6-9). As a matter of fact, this block is executed in class context before the actual class gets instantiated.

It receives the model, policy object and the params being passed to the operation call. How that all works and why the model is already available at this early point I will explain in a minute.

In the block you can run arbitrary code, inspect the incoming params, and so on to figure out what's the environment. `if/else` in a builder is required but it will be the only central place to differentiate between contexts.

In my block, I extensively make use of the policy object and call methods like `admin?` and `signed_in?` on it. These methods I had to add the the `Thing::Policy` class.

```

1 class Thing::Policy
2   # ...
3   def signed_in?
4     user.present?
5   end
6
7   def admin?
8     signed_in? and user.email == "admin@trb.org"
9   end

```

I add two methods to the policy. The `signed_in?` method accesses the `user` and makes sure it's present (line 3-5). Remember, both user and model get passed into the policy and are available at any point.

The `admin?` rule method checks if the signed-in user is an admin - in this example, this is a lame, hard-coded check of the user's email (line 7-9). You're free to run arbitrary logic here to figure out the user's role, I use the admin email for simplicity.

We'll get to the "*How did the user and model get into the policy?*" in the next section. Now that we understand how to query the policy for different environments, we can use that to build different operations.

```

1 builds -> (model, policy, params) do
2   return self::Admin if policy.admin?

```

In case of a logged-in admin user, the builder block will return `self::Admin`. The `self` namespace is a technique to dynamically evaluate a constant and it will make more sense when we get to the Update operation in a few pages.

Given our current environment, the `builds` block will return the constant `Thing::Create::Admin` for an admin user. Analogously to that we build a `SignedIn` operation for that particular case. If the block doesn't return anything, the original class `Thing::Create` is instantiated.

Note that the operation's main policy rule `create?` is not run at this point, the policy object is just instantiated for your convenience. The builder only works out what class to instantiate, the actual operation rule is run at a later point.

Resolver

Having the builder in place, you can play around with your operation to really understand how builders work. Since both new classes are empty, you can run the operation and it will perform the exact same thing for all three contexts. However, the operation's class will change.

```

1 Thing::Create().class           #=> Thing::Create
2 Thing::Create.(current_user: User.find(1)) #=> Thing::Create::SignedIn
3 Thing::Create.(current_user: User.admin)   #=> Thing::Create::Admin

```

Again I want to note how this changes things from the operation user having to know about contexts to a user simply passing arguments to the operation which, in turn, figures out internally what concrete class to instantiate.

We've created a strong interface while minimizing error sources.

This is called *polymorphism* as we will have different behavior without having to know what exactly is changing from the outside.

Now, to fully understand builders, we need to go a step back. In the Create class I included Resolver instead of Policy. A result of mixing in Resolver is that both policy and model are instantiated before the operation and therefore are available in the builder block.

Here's another pseudo flow snippet of what happens when you run the Thing::Create operation with Resolver being mixed in. It explains which steps are involved in order to give you a convenient setup for the operation dispatch.

```

1 Thing::Create.(params)
2   model      = build_model(params)
3   policy     = build_policy(params)
4   operation_class = build_operation_class(model, policy, params)
5   operation_class.new.(params)

```

1. The first thing, and that's different to the workflow we've learned before using Resolver, is that the model gets found or created on the class level (line 2), before the operation is instantiated. Other than that, CRUD semantics are identical.
2. Using the model and the params hash that got passed into the operation, the policy object is instantiated (line 3). Since the params hash contains the :current_user, the policy object has access to user and model and is ready to answer questions about rules.
3. Now, the builds block is run. You now understand where all the block arguments come from. The builder will always return a valid class constant (line 4).
4. This constant is then instantiated which results in an operation object that is being run (line 5).

The nice thing about builders is that we don't need to change our application code at all. This also applies to the tests. The caller of an operation shouldn't even know about the different subclasses, and therefore, our controller actions don't change at all.

```

1 class ThingsController < ApplicationController
2   def create
3     run Thing::Create
4     # ...

```

So far, we've created three operation classes for three different contexts: anonymous, signed-in, admin. However, they are still identical, so we should now start refining the signed-in and admin operations so our polymorphic dispatch actually makes sense.

Refining Operations with Modules

Earlier, I mentioned that the create and update form for `Thing` should have a "*I'm the author!*" checkbox when a user is signed in. This means we have to add a boolean field to the contract and we have to add logic to the operation that adds the current user to the freshly created thing if this box is checked.

Again, these are semantics for signed-in users, only, and must not be mixed with the operation for anonymous users. This is why I have different operations `Thing::Create` to handle anonymous users and `Thing::Create::SignedIn` for the registered counterpart.

Refining `SignedIn`'s contract could happen just in the particular class, as we did it in chapter 3. However, admin users should experience the same behavior, and we're also gonna need this for the Update operations, so I will put the refinement code into an *operation module* that sits at `app/concepts/thing/signed_in.rb`.

```

1 module Thing::SignedIn
2   include Trailblazer::Operation::Module
3
4   contract do
5     property :is_author, virtual: true, default: "0"
6   end

```

The separate file is a simple Ruby module in the `Thing` namespace (line 1). To allow using the operation API I include `Trailblazer::Operation::Module` (line 2). I can now refine my contract and all other composite objects of any operation.

As for the contract, all I do is adding an additional property `is_author` (line 5). It's a virtual field which won't get read and written to the model and defaults to zero. This in particular means: when the form is created for rendering or validation, this value will always default to zero.

The virtual field alone won't change anything in the including operations. To actually extend the behavior, too, I add a callback using the callback API in the operation module.

```

1 module Thing::SignedIn
2   # ...
3   callback(:before_save) do
4     on_change :add_current_user_as_author!, property: :is_author
5
6     def add_current_user_as_author!(thing, params:, **)
7       thing.users << params[:current_user]
8     end
9   end
10 end

```

Using an `on_change` event I register the new method `add_current_user_as_author!` in the `:before_save` callback group. According to the configuration, this method is only called when the `is_author` field has changed after validating the form (line 4). This is a very convenient way to add behavior to the operation without having to override any code.

The callback method receives the `thing` twin (actually, the contract, which is a twin) and also the `params` hash, so I can add the `:current_user` to the twin's `users` list (line 6-8). Since this is happening before we save, the added user object will be associated and persisted on the ActiveRecord level once we call `form.save`.

This code is beautiful, but it is still completely useless as we have to plug the new module into an actual operation to make it come into effect.

```

1 class Thing < ActiveRecord::Base
2   class Create < Trailblazer::Operation
3     class SignedIn < self
4       include Thing::SignedIn
5     end
6
7     class Admin < SignedIn
8   end

```

To enable the changes and additions, I include the new module into the `SignedIn` operation class (line 4). This will, at compile-type, extend the original class with the code from the module, add the property to the contract and push a new callback into the `:before_save` group.

Since `Admin` inherits from that class, nothing needs to be done there (line 7-8).

Polymorphic Testing

With Trailblazer, testing is fun! At least, that's what I keep telling myself. Since test files are getting bigger and bigger, I split them up into `test/concepts/thing/create_test.rb`, `update_test.rb`, and so on.

To assure that we didn't break anything and our `SignedIn` module does what we want, I add a new test block to `test/concepts/thing/create_test.rb`.

At the top of the test file, I add some fixtures we're gonna need throughout the tests.

```
1 class ThingCreateTest < MiniTest::Spec
2   let (:current_user) { User::Create.(user: {email: "fred@trb.org"}).model }
3   let (:admin)         { User::Create.(user: {email: "admin@trb.org"}).model }
```

Remember the policy that identifies admins via the special email address? I know it's sketchy, but to minimize redundancy I create this special user once (line 3).

To test the `:is_author` feature, I add a new describe block.

```
1 class ThingCreateTest < MiniTest::Spec
2   # ...
3   describe "I'm the author!" do
4     let (:user) { User::Create.(user: {email: "nick@trb.org"}).model }
5
6     # anonymous
7     it do
8       thing = Thing::Create.(thing: {name: "Rails",
9                               users: [{"email"=>user.email}], is_author: "1"},
10                             current_user: nil).model
11     thing.users.must_equal [user]
12   end
```

In the first test case, I make sure that `Thing::Create` doesn't process the `is_author` property. Even though it's set to "1" it doesn't crash (line 9). This is because we don't pass in a valid `current_user` which in turn will instantiate the "normal" `Thing::Create` operation and not one of the two subclasses that process this new property (line 10).

By testing that there's only one author - the user we passed in via `users:` - this test assures that our builder works and the original operation didn't get screwed up. To be honest, I probably wouldn't even test this since I trust Trailblazer and its builder pattern, but I thought it is a great way to demonstrate the power of polymorphism.

```

1 # signed-in
2 it do
3   thing = Thing::Create.(thing: {name: "Rails", users: [{"email":>user.email}],
4     is_author: "1"},
5     current_user: current_user).model
6   thing.users.must_equal [user, current_user]
7 end

```

The next case passes in an existing current user and hence instantiates `Thing::Create::SignedIn` (line 5). Given that `is_author` is a true value, I now require the operation to have two authors, the one I explicitly passed via the `users` fields, and the current user (line 6).

I added more tests with zero values for the `is_author` field to assure that those values are not processed and the current user won't get added. These tests are in the repository and very similar to the ones above.

The tests for admin semantics are identical. For the sake of understanding, here's another case I added for you, and not for the actual application as it tests redundant facts.

```

1 # admin
2 it do
3   op    = Thing::Create.(thing: {name: "Rails", users: [{"email":>user.email}],
4     is_author: "1"},
5     current_user: admin)
6   thing = op.model
7   thing.users.must_equal [user, admin]
8   op.must_be_instance_of Thing::Create::Admin
9 end

```

This time, it's the admin user that gets passed into the operation as current user (line 5). After testing that both users are added as authors I do something completely unnecessary (line 6). In the last line, I test if the actual operation is of type `Admin` (line 7).

Personally, I wouldn't write such an assertion as it tests internals. Anyway, say you had very complicated builder code and you don't want to test them via asserting side effects, you could also inspect the concrete operation class.

We've successfully finished the first chunk of work of this chapter. Dependent on the current user different operation subclasses get instantiated and expose differing behavior. Sharing generic logic happens via inheritance and modules in a very ruby-esque way. The tests only assert public semantics and do not know anything about the internals and how the desired result is achieved.

Polymorphic Views

After all this dry operation and test coding, let's put those changes into practice. It'll be cool to see different forms for the operations. Signed-in and admin users have to see that "I'm the author!" checkbox. Going further, I want the form to have an orange background if an admin is logged in.

When being confronted with a context-sensitive view that is supposed to change its apparel with different user types there's three options to tackle this problem.

We could simply use a Rails controller view with `ifs` and partials, and pass around locals and instance variables. We will have a hard time at some point figuring out what is the dependencies we need for the view.

Dependencies, that word is crying for view components. We could encapsulate the view in a polymorphic cell and use view inheritance to override parts of the template with user-specific content. We'd have a clearly visible interface. However, the view inheritance often produces more classes and view files than necessary.⁵¹

The third solution is combining the two above. Why not use a cell with some `ifs`, solving this problem of presenting a context-sensitive form in a pragmatic, yet clean way? Let's do that!

To convert the thing form into a cell, I literally rename the `app/views/things/new.html.haml` template and convert it to a cell view in `app/concepts/thing/views/form.haml`.

Here's the view top part after I "cellified" it.

```

1 %h3 What do you want to talk about?
2 = simple_form_for contract, html: { class: css_class} do |f|
3   = f.input :name, readonly: contract.readonly?(:name)
4   = f.input :description
5   = f.input :file, as: :file
6
7 %legend Do you know any authors?
8
9 - if signed_in?
10  = f.input :is_author, as: :boolean, label: "I'm the author!"

```

This view is identical to the old view, only that I changed the instance variable `@form` into `contract`. Also, I added a block adding the `is_author` checkbox (line 10). This field is only rendered when `signed_in?` is true (line 9). Again, I decided a simple `if` is more efficient than making it overly complex with view inheritance and a truely polymorphic view component. Know your tools, use them in moderation.

⁵¹In a future version of Cells there will be *view block inheritance* which allows overriding parts of the original view with template blocks being defined in the overriding cell itself. The result is a much simpler implementation without the problem of "too many files". I will speak about that in the *Cells Field Guide*.

After moving this view the controllers need to be updated. First, I add a new method `render_form` to `ThingsController`.

```
1 class ThingsController < ApplicationController
2   # ...
3   private
4   def render_form
5     render text: concept("thing/cell/form", @operation),
6            layout: true
7   end
```

I use the `concept` method to render the `Thing::Cell::Form` view model, something we still have to create (line 5-6). The cell's model is the operation object (line 5). Hereby, I clearly define a visible dependency.

The controller normally seeks to render a template file. Since we've sorted that using a cell, we need to use `render text:` to prevent the controller from being too passionate (line 5). As this render instruction will per default only display the HTML fragment of the cell, we need to pass `layout: true` to that call to embedd the cell nicely.

We can now use this method in all actions that are supposed to render the thing form. Here's how `ThingsController#new` now looks.

```
1 class ThingsController < ApplicationController
2   def new
3     form Thing::Create
4     render_form
5   end
```

Instead of the `render` statement you can now find the call to `render_form`. This almost completely bypasses the `ActionController` rendering stack and uses a cell for the content markup, instead.

I changed `create`, `update` and `edit` accordingly, please refer to the repository here.

Theoretically, the entire controller view is now handled by the new cell. The problem is: we still need to write that cell class. It goes to `app/concepts/thing/cell/form.rb`.

```

1 class Thing::Cell::Form < ::Cell::Concept
2   inherit_views Thing::Cell
3
4   include ActionView::RecordIdentifier
5   include SimpleForm::ActionViewExtensions::FormHelper
6
7   def show
8     render :form
9   end

```

A brand-new cell `Thing::Cell::Form` maps the entire controller view to a class. Normally, the cell would look up its views in `app/concepts/thing/cell/views`. In order to share the view directory with `Thing::Cell`, I instruct it to change its view path using `inherit_views` (line 2). The cell will now pick views from `app/concepts/thing/views`.

To satisfy Simpleform a few helpers need to be included manually (line 4-5). The main `show` method renders the `form.haml` view we just wrote (line 7-9).

Looking back into the `form.haml` view, you will see a couple of method calls that are undefined, yet. For example, `signed_in?` needs to be implemented. Let's write the rest of the cell.

```

1 class Thing::Cell::Form < ::Cell::Concept
2   property :contract
3
4   def css_class
5     return "admin" if admin?
6     ""
7   end
8
9   def signed_in?
10    model.policy.signed_in?
11  end
12
13  def admin?
14    model.policy.admin?
15  end

```

As we pass in the operation instance into the cell, I declare `contract` as a property (line 2). This will allow using `contract` in the view and delegates the call to `model.contract`, grabbin the form object from the operation.

Another method in the view is `css_class`. Only when `admin?` returns true, I add the CSS class `admin` to the form tag's classes (line 4-7).

In both view and cell code I extensively use `signed_in?` and `admin?` to differentiate between different contexts. Those methods are defined on the cell and literally just direct the call to `model.policy` (line 9-11 and 13-15). Since `model` is pointing to the operation instance, there must be a `Operation#policy` method defined.

This policy method does exist and returns the exact same policy object that we were using in our builder earlier (line 10 and 14).

Policies in Trailblazer are supposed to embrace all authentication-relevant code in a small-scoped object. They are also meant to be passed around, from the top-level operation down to the rendering cell or presenter. By giving operation users access to it, we're allowing policy-based decisions outside of the operation object while holding to the same rule-set that was used throughout the process.

Without being logged in, you will see the old, tedious form on `/things/new` that only gives you title, description and three author fields. Signed-in, you will observe an additional "*I'm an author!*" checkbox. As an admin, the whole form will be unreadable because of its bright orange background along with the same fields the signed-in user sees.

For different user contexts, we have three different semantics now. While this is pretty awesome, we can't go nuts just now. We should write functional tests verifying the different renderings and processings from the integrated request perspective.

Integration Test: Create

Even though we replaced the entire controller view with a cell, there's no need to write an isolated cell test. This global cell I will test on the functional level in integration tests.

We used to keep integration tests in one big `test/integration/thing_test.rb`. I split them up into separate assets to make it easier to navigate.

The `test/integration/thing/create_test.rb`. This is where we have test blocks for every function. Let's revisit this file. I am not going to explain every test as they have been covered in earlier chapters but will focus on new assertions.

I put the testing of functions like `new` or `create` into `describe` blocks. Within those blocks, `it` blocks will embrace assertions for one user type, resulting in a very clear test layout.

```

1 class ThingCreateIntegrationTest < Trailblazer::Test::Integration
2   describe "#new" do
3     # anonymous
4     it do
5       visit "/things/new"
6       assert_new_form
7       page.wont_have_css("#thing_is_author")
8       page.wont_have_css("form.admin") # no orange background.
9     end

```

The first to test is the new form page (line 2) in a describe section. We have discussed this before. However, I extracted several assertions into a method `assert_new_form` so we can reuse it for signed-in and admin user tests (line 6). Additionally, I make sure the form does not contain the “*I’m the author!*” checkbox (line 7). Also, the `admin` CSS class must not be set in this form as we are displaying for an anonymous, not signed-in user (line 8).

The `assert_new_form` method I put directly into the test class. It contains assertions from the earlier test and don’t need to be discussed, again.

```

1 class ThingCreateIntegrationTest < Trailblazer::Test::Integration
2   def assert_new_form
3     page.must_have_css "form #thing_name"
4     page.wont_have_css "form #thing_name.readonly"
5     page.must_have_css("input.email", count: 3)
6   end

```

We basically mimic a quick browser test using our smoke test and make sure the form to create things works. Now, the same for a signed-in user.

```

1 class ThingCreateIntegrationTest < Trailblazer::Test::Integration
2   # signed-in.
3   it do
4     sign_in!
5     visit "/things/new"
6     assert_new_form
7     page.must_have_css("#thing_is_author")
8     page.wont_have_css("form.admin") # no orange background.
9   end

```

This is almost identical to the anonymous test except for two things. I sign in using the familiar `sign_in!` method (line 4). Since we’re logged in, I make sure the checkbox is visible (line 7).

Another `it` block will test the admin form in a similar way.

Lifecycle Smoke Test

Of course, testing the form rendering, only, is not enough. Next challenge is to test the creation lifecycle of things.

Say you were testing this manually. You'd click through the form with valid and invalid data, correct it, resubmit, check the box, and so on. Eventually, you would sign in as a user, do the same again, and then again as an admin. Tedious, but you can make sure all contexts work.

We do just this with a lifecycle smoke test. Just imagine you transcribe your manual click test to a Capybara test. Again, the `describe` block embraces the whole lifecycle test whereas `it` blocks cover different user roles.

```
1 class ThingCreateIntegrationTest < Trailblazer::Test::Integration
2   describe "click path" do
3     # anonymous.
4     it do
5       visit "/things/new"
6       # create thing ..
7       page.current_path.must_equal thing_path(Thing.last)
8       page.wont_have_css "a", text: "Edit"
9     end
10    end
```

As an anonymous user, I try to create an invalid thing, assert the rendering of an invalid form, and so on. We already tested this. However, I added one more test where I make sure the “Edit” link is not rendered for a random user (line 8). Remember, this is only done for authors from now on.

Now, a bit more complex for a signed in user.

```
1 describe "click path" do
2   # signed-in.
3   it do
4     sign_in!
5     visit "/things/new"
6
7     # invalid.
8     click_button "Create Thing"
9     page.must_have_css ".error"
10
11    # correct submit.
12    fill_in 'Name', with: "Bad Religion"
13    check "I'm the author!"
14    click_button "Create Thing"
```

```

15  # /things/1
16  page.current_path.must_equal thing_path(Thing.last)
17  page.must_have_content "By fred@trb.org"
18
19  # edit
20  click_link "Edit" # /things/1/edit
21 end

```

Our `sign_in!` helper makes me a valid, logged-in user on the HTTP level (line 4). I then test the submission of incorrect data, then correct data (line 8-12). To go completely nuts, I check the new checkbox (line 13). This will implicitly make sure this box is actually there, so I don't need to write another CSS test for that.

I submit the form and verify that the signed-in user is now the author (line 17). In another implicit test, I click the “*Edit*” link. This would, again, raise an exception if that link wasn't there and is the perfect way to make sure the cell renders the correct links following to the policy object's dictate.

The admin click path test is similar and can be admired in the repository.

We implemented some new, refined operations for different user contexts and tested those in operation unit tests. The wiring into the UI is thoroughly asserted using our smoke tests. Now that we know that creating of things works for all three user types, we can move on to editing, updating and deleting things with policies.

Updating

The `Thing::Update` operation we implemented over a few chapters sits in `app/concepts/thing/update.rb`. So far, this is only one class inheriting from `Create` that has a slightly changed contract along with some changed post-processing logic allowing to delete authors from a thing, and so on.

Analogously to the `Create` structure, I will now introduce `Update::SignedIn` and `Update::Admin` to embrace the three different contexts we have.

Here's the new class header for the basic update operation that applies to not signed-in users.

```

1 class Thing < ActiveRecord::Base
2   class Update < Create
3     self.builder_class = Create.builder_class
4     action :update
5     policy Thing::Policy, :update?

```

Inheriting from `Create` will copy contract, policy, representers and callback objects to the new class.

Builders in Trailblazer are not inherited, though. Manually, I have to reference `create`'s builder block using `builder_class` (line 3). The main thought behind that is in most cases when using operation

inheritance, you do not want to replicate builders in subclasses as the building should only happen once.

The CRUD module in this operation is supposed to find the correct model for us, which is why I set `:update` as the action. We've discussed that in earlier chapters (line 4).

To make `Update` use the correct policy, I configure `update?` as the new rule that is evaluated when the operation is run (line 5). Here's the new rule in `app/concepts/thing/policy.rb`.

```
1 class Thing::Policy
2   def update?
3     edit?
4   end
5
6   def edit?
7     signed_in? and (admin? or model.users.include?(user))
8   end
```

In our policy class, I map `update?` to `edit?` for consistency as I like to make updating operations use the `update?` rule, whereas in views, the `edit?` rule feels more intuitive (line 2-4). They are identical, though.

The `edit?` rule is valid if, and only if, there is a current user being passed into the operation and if that particular user is either an admin or is an author of the `Thing` we're trying to update (line 6-8).

Policy and Models

It is important that we encapsulate this kind of logic in a policy object, and it is totally fine to use model-specific code here. The policy is exactly the place for decisions about environments identified by different user types in combination with decisions about model state.

Going further, the code in the policy object is not to be replicated anywhere else. If you need rule-based decision code, you have to use policy objects in Trailblazer. This is why the operation makes it easy to access its policy object using the `policy` method. We've seen one example already where that is being used outside of the operation: in the cell we built in the last section.

That being said and implemented, the `Update` action will now raise a policy breach whenever it is run with a not signed-in, anonymous user. We will test that shortly.

Composable Operations

To have a consistent structure of operations, we want to introduce `Update::SignedIn` and `Update::Admin` now. The following snippet shows the code for the operation handling the signed-in case in `app/concepts/thing/update.rb`.

```

1 class Update < Create
2   #
3   class SignedIn < self
4     contract do
5       property :name, writeable: false
6       # here is the rest of the original code from Create...
7     end
8   end

```

I simply moved the original class code from `Update` in the earlier chapters into the `SignedIn` class. To illustrate that I added some of that code to the above snippet (line 4-7). In other words: The old `Update` is now `Update::SignedIn`.

`SignedIn` inherits from `Update` exactly as we did it with `Create` and its subclasses before. This inherits the contract and all other parts. The contract I added makes the `name` field read-only and adds the “*Remove!*” author function. We discussed all that in earlier chapters.

However, the crucial change here is, and of course you’ve noticed that already: the `SignedIn` operation does not include the `Thing::SignedIn` module which adds the “*I’m the author!*” field and its associated logic to the. Why do I skip this for this operation?

Well, the answer is quite simple. This particular operation class only handles signed-in users. Per policy, signed-in users can only edit things they authored - there simply is no need to have this checkbox in this context, so I skip including this module.

This is a helpful example of how operation modules let you compose operations for different contexts. In this case, the `Update::SignedIn` simply doesn’t need this function, it’s the other way round: if we had this field, we would allow users to add them multiple times and we would have to throw in additional logic to protect them from doing so. Instead, I simply do not include this module which will result in an operation that doesn’t process the `is_author` field.

It might be easier to discuss the opposite case now. The `Update::Admin` operation is only used for admin users. That means an admin user can edit any record, whether they are author or not, and hence the `is_author` checkbox does make sense in this context.

This is why I include the module for the `Admin` operation.

```

1 class Admin < SignedIn
2   include Thing::SignedIn
3
4   contract do
5     property :name
6   end
7 end

```

By deriving it from `SignedIn` we inherit all configuration and code (line 1). As I do want the author checkbox, I include the `Thing::SignedIn` module that we wrote in the beginning of this chapter (line 2). This will simply add the new field to the contract and import the additional business methods.

The last change is to make the `name` field writeable. While normal, signed-in users can't change the name of things, admins can. I simply override the original property without the `writeable` option (line 5).

Ok, this was a lot of inheritance and composition. What do we actually have? Three operations that do slightly different things.

1. `Thing::Update` was only introduced for consistency. This operation inherits all code from `Create`. The builder is copied, too. It keeps the policy configuration and the CRUD-specific setting. `Update` can be called but will always bail out with a policy exception making it a virtual operation with declarative value, only.
2. `Thing::Update::SignedIn` inherits from the latter, adds contract configuration and code specific to signed-in users which allows them to edit the description, and add or delete authors, only.
3. `Thing::Update::Admin` inherits all of that, adds the "*I'm the author!*" checkbox and its process code by including `Thing::SignedIn`, and makes the `title` field writeable.

As the root operation `Update` contains the builder configuration, controllers can remain as they are. The correct subclass will be instantiated internally in the operation.

Contract-specific Views

In the create and update form, we no longer can ask the policy whether or not to display the `is_author` checkbox. While we used to find that out by asking `policy.signed_in?` this doesn't apply to our requirements anymore. Even when signed in, this field should not be shown in the cases we discussed.

It is best to make the view template not know about internals, anyway, so I change `form.haml`.

```

1 - ...
2 %legend Do you know any authors?
3 - if has_author_field?
4   = f.input :is_author, as: :boolean, label: "I'm the author!"
```

The query method is now `has_author_field?` and “verbally” decouples the view from the internal implementation (line 3). This method needs to be implemented in `Thing::Cell::Form` now.

```

1 class Thing::Cell::Form < ::Cell::Concept
2   def has_author_field?
3     contract.options_for(:is_author)
4   end

```

And now, listen up. We've done this before, but what I use here does highly couple the view to the contract. I use `options_for` on the contract object to find out whether the operation's contract does have a `is_author` field (line 3). If so, the cell will render the visual counterpart.

Relying on contract internals can be dangerous and might lead to unexpected results. In my case, I find this totally fine, though. I know that the operation's contract will reflect the policy settings, and I am aware that I am passing an operation instance into the cell. All I do now is using part of the public APIs of operation and contract and combining it with knowledge about the operation's contract semantics.

To simplify that, you could expose a reader `has_author_field?` directly on the operation, which would make the operation part of the presentation. Or you could introduce a context object that reflects those settings. As Trailblazer is still quite young, both solutions are subject to experimenting and the emerge of best practices.

You may now sign in as different users, and you will see: the form will only show the `is_author` field for admins. Nice work. Now, tests!

Update Tests

Luckily, we wrote a bunch of tests for Update when we implemented it back then, without the concepts of user roles, though. I left those tests as they are.

In the top, I define some factories for different user types.

```

1 class ThingUpdateTest < MiniTest::Spec
2   let (:current_user) { User::Create.(user: {email: "fred@trb.org"}).model }
3   let (:admin)        { User::Create.(user: {email: "admin@trb.org"}).model }
4   let (:author)       { User::Create.(user: {email: "solnic@trb.org"}).model }

```

Using the `User::Create` operation several user roles are set up (line 2-4) which will come in handy in the tests.

Again, I structure the test file `test/concepts/thing/update_test.rb` with `describe` blocks for the different user roles. I start with anonymous.

```

1 class ThingUpdateTest < MiniTest::Spec
2   describe "anonymous" do
3     let (:thing) { Thing::Create.(thing: {name: "Rails", description: "MVC", \
4       "users"=>[{"email"=>author.email}]).model }
5
6     it do
7       assert_raises Trailblazer::NotAuthorizedError do
8         Thing::Update.(
9           id: thing.id,
10          thing: {name: "Rails", description: "MVC"})
11        end
12      end
13    end

```

The first let defines a thing with an author (line 3-4). I run the Update operation without passing in a :current_user, though (line 8-10). As the operation's policy will detect that breach I catch and assert that using assert_raises (line 7 and 11).

If I wanted, I could add more tests where the thing doesn't have authors, and so on, but I trust my super simple policy object and assume this is enough to verify the security works for anonymous users.

For signed-in users, I write more tests as the policy is a bit more complex here. I will omit some of them as they are conceptually identical: Test if the operation raises a NotAuthorizedError when run with the incorrect signed-in user.

I define a thing factory as I reuse this particular combination several times.

```

1 describe "signed-in" do
2   let (:thing) { Thing::Create.(thing: {name: "Rails", description: "MVC", \
3     users: ["email"=>author.email] }).model }

```

This is the same factory we used for the anonymous user. However, I still copy it to the new describe block (line 2-3). Every block embraces a separate test environment, and I find it simpler to just copy factories and then later, when you change one, you don't break the other blocks, instead of trying to be super-DRY and then run into problems and having to figure out factory dependencies.

Here's a test for a valid processing the Update operation with a signed-in user.

```
1 it "persists valid, ignores name, ignores is_author" do
2   Thing::Update.(  

3     id:           thing.id,  

4     thing:        {name: "Lotus", description: "MVC, well..", is_author: "1"},  

5     current_user: author).model  

6  

7   thing.reload  

8   thing.name.must_equal "Rails"  

9   thing.description.must_equal "MVC, well.."  

10  thing.users.must_equal [author]  

11 end
```

In the `Update` call, I use `thing.id` which will make sure the actual `Thing` row gets created before getting updated (line 3). I then pass in the current user which is, according to our factory, also the author of this model (line 5). Being lazy, I test two more things in this test case: I pass in `:is_author` and `:name`, too (line 4). In a real project I would probably write separate test cases illustrating this.

For the assertion part I test that all that was changed was the `description` field, all other arguments must be ignored by the operation (line 9-11).

Since the admin tests are very similar, I will save this room for more interesting things.

Testing Side-Effects

We've tested many things in few, concise and intuitive test cases. We know the builders work. However, we didn't write explicit builder tests which would equal to testing private internals. Instead, we verified this works by testing the implicit side-effects which will allow us refactoring more easily.

By injecting unsolicited parameters in combination with different user roles, we can make sure the correct operation is instantiated leveraging the correct contract.

The nice thing is: we also tested the proper working of our operations at the same time. This saves us from writing complicated unit tests while exposing knowledge about the operation's architecture.

As for the smoke tests I will limit myself to pure rendering assertions to make sure forms get rendered the way the current user should see it.

The tests for to edit form rendering go into `test/integration/thing/update_test.rb`. In the first case, the policy breach is tested where an anonymous user tries to access an edit form.

```

1 class ThingUpdateIntegrationTest < Trailblazer::Test::Integration
2   describe "#edit" do
3     # anonymous
4     it "doesn't work with not signed-in" do
5       visit "/things/#{thing.id}/edit"
6       page.current_path.must_equal "/"
7     end
8   end

```

In good Trailblazer manner one describe block per function (line 2-8). Similar to the create tests I have a let (:thing) factory defined in the test which you can look up in the repository. I then simply visit this thing's edit path (line 5). Since we're not signed in, I'm redirected to the root page (line 6). I will speak about this redirect after the tests are green.

Analogously, I test this case for the signed-in author of this particular thing.

```

1 describe "#edit" do
2   # signed-in
3   it do
4     sign_in!
5     visit "/things/#{thing.id}/edit"
6     page.wont_have_css "form.admin"
7     page.must_have_css "form #thing_name.readonly[value='Rails']"
8     page.must_have_css "#thing_users_attributes_0_email.readonly[value='fred@trb\
9 .org']"
10    assert_edit_form
11  end

```

Sign-in and browsing to the form is now something we've discussed sufficiently (line 4-5). I make sure the orange background is not present by asserting the absence of the .admin CSS class (line 6). Then I test the form is present, the title field is readonly, and other tests that I extracted into assert_edit_form the way we've done it for create. We went through all these tests earlier.

Likewise, the admin it block bores me and doesn't need to be printed here as it mostly identical to the above test. One thing worth mentioning is that I explicitly test that the title field now is readonly with the following assertion.

```
1 page.wont_have_css "form #thing_name.readonly"
```

This is a weak test, but so far, I haven't found out how to find an element and then make sure it does *not* have a certain CSS class. Anyway, this is also covered in the lifecycle smoke tests that are the last thing to do for this section.

Instead of repeating the lifecycle again in a different test file, I put additional tests into `test/integration/thing/create_test.rb`.

This snippet shows the extended click path test of a signed in user working on a `Thing`. “ruby class `ThingCreateIntegrationTest < Trailblazer::Test::Integration` describe “click path” do `sign_in! # .. old test code that leads to things/1 # edit click_link “Edit” # /things/1/edit page.must_have_css “form #thing_name” page.must_have_css “form #thing_name.readonly”`

```

1  # update
2  fill_in "Description", with: "Great!"
3  click_button "Update Thing"
4  page.current_path.must_equal thing_path(Thing.last)
5  page.must_have_content "Great!"

end ""

```

This goes into the original `it` block that represents the signed-in lifecycle. After we arrive on the `thing`'s show page, I click the “`Edit`” link and assure the edit form is there (line 6-8). Filling out the description field and submitting the form represents the new part of the test (line 9-10). After the submission we should be back on show page of the `thing` (line 11). Given our logic works, the description should now be updated (line 12).

A similar test for admin is in the same `describe` block.

Our entire logic and rendering is now tested with all three possible user roles. Time to celebrate! But wait, when a policy exception is raised, how does that redirect to the root page? This code sits in `ApplicationController` and, again, is stolen from Pundit.

```

1  class ApplicationController < ActionController::Base
2    rescue_from Trailblazer::NotAuthorizedError, with: :user_not_authorized
3
4    def user_not_authorized
5      flash[:message] = "Not authorized, my friend."
6      redirect_to root_path
7    end

```

The `rescue_from` method allows to configure a declarative handler if a `NotAuthorizedError` is encountered (line 2). The `user_not_authorized` method to handle this is purely HTTP-related and hence not encapsulated in an operation. It redirects to the root page and notifies the user with a flash message (line 4-7).

Delete

After having implemented all those nifty features to add and edit things the database keeps growing and growing with test entries. As I am a minimalist when it comes to technical experiences my 4-year old laptop keeps getting slower with every minute. We need a delete function to relieve my jurassic hardware!

This almost is a no-brainer. `Thing::Delete` goes into `app/concepts/thing/delete.rb`.

```
1 class Thing::Delete < Trailblazer::Operation
2   include CRUD
3   model Thing, :find
4   policy Thing::Policy, :delete?
```

And here, I am breaking my own conventions. Even though we introduced this best practice of having one class per function and user role, our delete will do the very same for signed-in and admin, and it will simply raise a policy exception for anonymous ones as they can't delete anything.

Note that I did *not* include the Resolver here as we don't have a builder block.

It would add a lot of classes, builder code and indirection for the sake of sticking to the convention. This is not necessary here, and you will soon see why.

I do inherit from the `Operation` class and not `Create` which is why I need to include `CRUD` and redefine the `model` configuration (line 1-3). Again, inheriting from one of the other `CRUD` classes would introduce a lot of unnecessary assets. I prefer to have a clean class for simple things.

The policy rule is `delete?` and will be the next thing to implement (line 4).

```
1 class Thing::Policy
2   # ...
3   def delete?
4     edit?
5   end
```

That was easy. I simply map the `delete?` rule to `edit?`, which we defined earlier. To refresh your memory that's already stuffed with polymorphic concepts, contract fields and what not: the `edit?` rule checks if the user is an admin, and also allows signed-in users to delete if they are author of the `thing` model. And this policy is exactly what I want for `delete`, too.

Here is the process part of `Thing::Delete`, and after that we will learn how that all plays together and why we don't need a builder here.

```

1 class Thing::Delete < Trailblazer::Operation
2   # ...
3   def process(params)
4     model.destroy
5     delete_images!
6     expire_cache!
7   end

```

The main task of this operation is to delete the model, or, in ActiveRecord speak: destroy it (line 4). As post-processing logic, I add dispatches to `delete_images!` and `expire_cache!` (line 5-6).

As you have guessed already, `delete_images!` will physically remove the uploaded files.

```

1 def delete_images!
2   Thing::ImageProcessor.new(model.image_meta_data).image! { |v| v.delete! }
3 end

```

I reuse the handy `ImageProcessor` class, pass in the `thing` instance's `image_meta_data` hash and use Paperdragon's `image!` processor which allows me to unlink all attached files with the `delete!` method (line 2)⁵².

```

1 class Thing::Delete < Trailblazer::Operation
2   include Gemgem::Cell::ExpireCache

```

The `expire_cache!` method is imported from the module we already built (line 2).

Unit Test: Delete

It's probably best to discuss the overall function of `Delete` while we go through some unit tests in `test/concepts/thing/delete_test.rb`

```

1 class ThingDeleteTest < MiniTest::Spec
2   let (:current_user) { User::Create.(user: {email: "fred@trb.org"}).model }
3   let (:admin)         { User::Create.(user: {email: "admin@trb.org"}).model }

```

I define several factory methods of user types we're gonna need throughout the test (line 2-3).

In this test file, I structure by thing states where the first block is to test against a thing without any authors. I wouldn't add this test in a real app, as our policy is thoroughly tested already but it is great to demonstrate the operation policy.

⁵²I am reusing the `ImageProcessor` that I extracted from the `Thing::Contract` in a [separate branch](#).

```

1 describe "authorless" do
2   let (:thing) { Thing::Create.(thing: {name: "Rails"}).model }
3   # anonymous
4   it "can't be deleted" do
5     assert_raises Trailblazer::NotAuthorizedError do
6       Thing::Delete.(id: thing.id)
7     end
8   end

```

In the first test, I run the `Delete` operation with an existing thing id (line 6). I omit passing in a user here, which makes this an anonymous context. Given our operation policy `delete?` works, this should raise an exception and not run the operation (line 5-7).

I do the same for a signed-in user and for admin. The latter will work, because admins are allowed to delete just anything. The signed-in case will raise an exception, too, as the user is not an author.

The following test case block is for a thing with an author. Most of the test cases verify that only authors and admin can delete things.

Here's how I test that the images of a thing actually get deleted, too.

```

1 describe "with authors" do
2   it "deleted by author, with images and comments" do
3     thing = Thing::Create.(thing: {
4       name:      "Rails", is_author: "1",
5       file:      File.open("test/images/cells.jpg")},
6       current_user: current_user).model
7
8     thing = Thing::Delete.(id: thing.id, current_user: current_user).model
9     thing.destroyed?.must_equal true
10    Paperdragon::Attachment.new(thing.image_meta_data).exists?.must_equal false
11  end

```

The test setup is a thing with the current user being author, and it comes with an uploaded image, too (line 3-6). After calling invoking the delete operation, I assert the model actually got destroyed (line 8-9).

Using the `Attachment` class manually, I make sure the files were physically unlinked by calling the `exists?` method (line 10).

Binary Operation Context

Why didn't we need a builder in this operation? Think about what the operation actually does. We have a simple, binary setup here: the operation either shouldn't be run at all, or it will do the exact

same for any context. The operation doesn't care if it's an admin or an author deleting, it will bust the model along with its assets.

The only problem is to figure out whether or not to run, and this is exactly what the operation's policy does! Since the policy is the first stop when invoking an operation, we can authorize the call or raise a policy violation before anything else is run.

To delete things from the web UI I will insert the “Delete” link right after the “Edit” link on the thing's show page in `app/views/things/show.html.haml`.

```

1 %h1 #{@thing.name}
2   - if @op.policy.(:edit?)
3     = link_to "Edit", edit_thing_path(@thing)
4   - if @op.policy.(:delete?)
5     = link_to "Delete", thing_path(@thing), method: :delete

```

The link for deletion is completely analogue to the edit one, except for instructing Rails to fake a DELETE request (line 5). This will route the click to `ThingsController#destroy` without having to add routes. Note that I reuse the `delete?` rule on the policy object to find out if we should render this link (line 6).

I feel its almost unnecessary to discuss the `destroy` action in the controller, but here is it anyway.

```

1 class ThingsController < ApplicationController
2   def destroy
3     run Thing::Delete do |op|
4       flash[:notice] = "#{op.model.name} deleted."
5       return redirect_to root_path
6     end
7   end

```

If the `Delete` operation is run successfully, a flash message will inform the user and they are redirected to the homepage (line 3-6).

Tests for the wiring all go into the lifecycle tests in `test/integrartion/thing/create_test.rb`. In the anonymous block I make sure the “Delete” link is not rendered, right after I tested the same for editing.

```

1 class ThingsControllerCreateTest < Trailblazer::Test::Integration
2   describe "#new" do
3     # anonymous
4     it do
5       visit "/things/new"
6       # ...
7       page.wont_have_css "a", text: "Delete"
8     end

```

By asserting the delete link is absent, I know the policy decider in the view works (line 7).

I also test, for both signed-in and admin, if I can click the delete link and if that redirects me. The extended admin case looks as follows.

```

1 # admin.
2 it do
3   sign_in!("admin@trb.org")
4   visit "/things/new"
5   # ...
6   click_link "Delete"
7   page.current_path.must_equal root_path
8 end

```

The last two lines implement just that. I click the link and hope I get redirected (line 6-7).

Our application now allows saving, with is another reason to throw another party! We tested all this new behavior and can be sure it works.

Impersonate

However, when writing the example app and playing with ideas, I benefited a lot from one Tyrant feature. The ability to impersonate other users. In other words, when being logged-in as admin, you can view pages as if you were another user.

This is the last feature for this chapter and it will greatly help you when experimenting with all the learnings about polymorphic operations and policies as it makes it super-simple to simulate other users browsing the app.

To activate impersonation all I have to do is change some lines in ApplicationController.

```

1 class ApplicationController < ActionController::Base
2   before_filter { Tyrant::Session::Impersonate.(params.merge!(tyrant: tyrant)) }

```

I replaced the call to `Tyrant::Session::Setup` with `Impersonate` (line 2). This does exactly what the old call does, with the exception if the key `:as` exists in the `params` hash.

Given we were browsing to the URL `/things/new?as=jimmy@trb.org` Tyrant will now find the new user by its email and replace `params[:current_user]` with the found user. After that, a new key `params[:real_user]` is set to the original user.

That is all that happens. When passing params to other operations now, they will see the new `:current_user` value and run the respective code for that role. It's a bit as when you unit-test your operations and pass in different users.

The important thing here is: Tyrant doesn't even bother to change any global variables. In Trailblazer, there are no global variables. Operations, view models, and representers will always receive an explicit `current_user` object or a policy instance that was created before, again, in an operation. There is no need to magically reset globals after the request is run, the way it is done in other authentication frameworks. We simply change the `params` hash locally.

You can test this now and you will see that you can view pages as any kind of user, even if you're not an admin. That's why we should throw in our own `Impersonate` implementation in `app/concepts/session/impersonate.rb`.

```
1 class Session::Impersonate < Tyrant::Session::Impersonate
2   include Policy
3   policy Thing::Policy, :admin?
4 end
```

This new operation simply inherits all behavior from Tyrant (line 1). By adding a policy that uses the familiar rule `admin?` I make sure no-one but admin can run this (line 2-3). In the Tyrant implementation, the original `Setup` is run before the policy is evaluated, allowing everyone to get authenticated, but not to impersonate.

To activate it, we have to change the application controller.

```
1 class ApplicationController < ActionController::Base
2   before_filter { Session::Impersonate.(params.merge!(tyrant: tyrant)) }
```

I replaced the operation dispatch with our own implementation (line 2).

When signed-in as a non-admin now, you will be kicked to the homepage when trying to impersonate.

The way Tyrant operations can be customized just by subclassing and extending or overriding parts using pure Ruby is what I love about the Trailblazer architecture. Imagine you had to change behavior in Devise - this becomes a nightmare once you need to move beyond the declarative interface due to the lack of clean encapsulation of the functions and its hardcore-coupling to Rails in combination with global variables.

NavigationCell and Impersonate

As promised in the beginning of this chapter, we will now add some more logic to the `Navigation::Cell` to make it render differently when you are actually using this feature. Being impersonating someone else, the tooltip on a star icon next to the welcome message should say “*You really are: admin@trb.org*”.

To do so, we need to extend the arguments we pass into the navigation cell in `app/views/layouts/application.html.haml`.

```
1 %header
2   = concept "navigation/cell", OpenStruct.new(
3     real_user:      params[:real_user],
4     current_user:   params[:current_user],
5     "signed_in?" => tyrant.signed_in?)
```

The real user is now passed into the cell, too (line 3). As the session setup will always populate the `:real_user` I do not have to worry about anything here.

In the cell code, a little change will bring a cool, very helpful feature.

```
1 module Navigation
2   class Cell < ::Cell::Concept
3     property :current_user
4     property :real_user
```

I add the `real_user` property so we have convenient access to it in the cell (line 4).

Then, I extend the `welcome_signed_in` method that renders the hello message.

```
1 def welcome_signed_in
2   link_to("#{impartate_icon} Hi, #{current_user.email}".html_safe,
3         user_path(current_user))
4 end
```

As you can see, I simply added a call to `impartate_icon` that will render the necessary markup (line 2).

The implementation is basic cells logic.

```

1 def impersonate_icon
2   return unless real_user
3   "<i data-tooltip class=\"fi-sheriff-badge\""
4   title=|"You really are: #{real_user.email}"|"></i>"
5 end

```

The method won't return anything unless `real_user` is present. So, in case this is nil, nothing additional is rendered (line 2). I then manually compose an HTML string where I use a Foundation icon with a tooltip revealing the real users email address (line 3-4).

Testing Impersonation

I know exactly that you already tried this feature right away, without any tests. This is alright, but let's finish this chapter with a quick functional test in `test/integration/session/impersonate_test.rb`

```

1 class SessionImpersonateTest < Trailblazer::Test::Integration
2   let (:jimmy) { User::Create.(user: {email: "jimmy@trb.org"}) }
3   before { jimmy }
4
5   it do
6     visit "/?as=jimmy@trb.org"
7     page.must_have_css "a", text: "Sign in" # not logged in.
8   end

```

As always, a test user is defined in the test header (line 2). To create it explicitly, I use `before` so it is around in every test case (line 3). When trying to use the impersonate feature as an anonymous user, you won't be lucky. The system will still ask you to log in (line 6-7).

A similar test I write for a signed-in user.

As a last test case I login as an admin.

```

1   it do
2     sign_in!("admin@trb.org", "123456")
3     visit "/?as=jimmy@trb.org"
4     page.must_have_content "Hi, jimmy@trb.org"
5   end

```

After signing in, and using the `as` key, the system must welcome us as a different person (line 2-4).

And that's it, we have a closed system now that only exposes a subset of its functionality to anonymous users. Implementation of contexts is not done with deciders spreading all over the layers of the framework, but cleanly separated in different classes.

By using inheritance between classes that implement the same function, but different user roles, we can easily share contracts and policy objects without the dreaded "black-hole inheritance" that you were probably thinking of when you started reading this chapter.

And if you don't like inheritance, you are free to use Trailblazer's compositional features where you can copy and inherit parts of other operations, as we did in several places in this chapter.

After operations have been run, the presentation layer can leverage policy objects to reflect restrictions visually, again, without any redundancy as the policy object is passed between the layers.

The operation users don't even know there is different subclasses implementing the requirements. By applying builders to the top class, the polymorphic dispatch happens internally. This was also beneficial in our tests. Remember, we didn't test any internals, only the side-effects on the application state.

This is applied object-orientation in a real system, but without the pain that Rails makes you think there is.

In the next, two final chapters we will learn how to build a document-based HTTP API on top of what we've developed. After reading that, you're more than ready to jump on your own projects and apply Trailblazer like a pro!

Hypermedia APIs: Rendering

When writing modern applications, it often is not enough to provide a neat user interface that we can leverage by clicking buttons in a web browser. We've come to the conclusion that applications must expose more than one way for being operated.

Letting machines, or real users, query and modify application state by pushing documents back and forth between endpoints, is a fantastic architectural approach which can be added to an existing system, or be used to physically separate an highly coupled, complex software.

Adding API code on top of existing HTML-based user interface implementation gives you two options.

One way would be to mix your new code into the old code. Most of the merging would happen in controllers, as they act as endpoints for HTTP. This would mean you have to render HTML and JSON in the same controller, and differentiate between the two formats.

Rails tried to implement this using responders and leveraging a clumsy format DSL in controllers. Both are extremely unpopular - an indicator that it's not a good idea to mix concerns here.

The other way is pretty much what we've done so far. For different environments, introduce different objects to handle. In other words: leave your HTML code where it is and add additional "API" controllers to process document requests.

And as you already have guessed, my dear reader, we're gonna choose the second way. Now that we have different operation classes to handle all the various environments, why would we intermingle user interface and API code?

Code for the following two chapters can be found [in our repository⁵³](#).

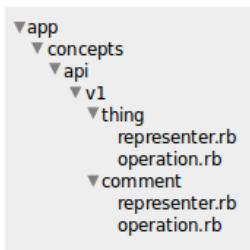
File Structure

After having played a lot with different file layouts, I decided it's best to organize API-specific code in a completely separate directory while still following the Trailblazer convention.

I think about the API code as a completely different component in our application, a different component that could even be a separate application.

The dependencies are one-directional, where the API code will use the old code from the last chapters, but the operations and contracts and whatnot we've written so far won't be changed, they won't even know there's API code depending on it.

⁵³<https://github.com/apotonick/gemgem-trbrb/tree/chapter-11>



So it's best to put that API code into its own directory.

The way we organize code for the API is straight-forward. We will introduce a new super concept `app/concepts/api`, add another level of separation, the folder `v1` and then save our operations and representers in this directory.

I introduce the `v1` directory and namespace to make it easier to push out new versions of our API. In this chapter, fortunately, we're only going to write one version. Nevertheless, API code, which is a publicly used concept, should be organized by versions. This is not me speaking but the opinion of a bunch of very experienced API developers I've interviewed exclusively to provide you the lastest best practices in this read.

Rendering Comments

As a light-weight introduction to representers and Trailblazer, why not start with a simple render-only endpoint? Let's implement a controller action that sits on the route `/api/v1/comments/:id` and delegates the JSON serialization to a Trailblazer operation? Does that sound good?

When writing API code, it can get quite hard to test your code. You could install the Postman extension and issue requests against your endpoints from it, and then check the headers and JSON bodies manually.

But why do that? It's awkward, tedious and error-prone. What I usually do is I start with an integration test. And this is what we do right now.

The API integration test I put into `test/api/v1/comments_test.rb`. In the file name, I am reflecting the URL which makes it very easy to find tests for endpoints.

```

1 class ApiV1CommentsTest < MiniTest::Spec
2   include Trailblazer::Test::Integration
3
4   describe "GET /comments/1" do
5     let (:thing) { Thing::Create.(thing: {name: "Rails"}).model }
6     let (:comment) do
7       Comment::Create.(
8         id:      thing.id,
9         comment: {
10           body: "Love it!", weight: "1", user: { email: "fred@trb.to" } }
11       ).model
12     end
  
```

In the comments integration test, I use `describe` blocks to structure the different endpoints, so I have test blocks for rendering, creating, updating, and so on, in line with our test layout for pure operations.

First, I setup the application state by creating a thing and a comment. As always, operations help me to instantiate a realistic scenario, and I use both `Thing::Create` and `Comment::Create` with the well-known syntax (line 5-12).

The actual test is pretty much self-explaining.

```
1 describe "GET /comments/1" do
2   it "renders" do
3     get "/api/v1/comments/#{comment.id}"
4
5     last_response.body.must_equal(
6       {
7         body: "Love it!",
8         _links: { self: { href: "/api/v1/comments/#{comment.id}" } }
9       }.to_json
10    )
11  end
```

Calling `get` with a provided path will literally send a `GET` request to that endpoint. This happens without hitting the network, but there's no mock magic involved to fire that request.

By including the comment's id, creation of the comment and the associated thing is assured (line 3).

To verify the returned document, I use `last_response.body` and really just compare it to a string (line 5-10). The string is constructed by defining a hash and then transforming it to JSON by calling the global `to_json` method (line 6-9). I do this mainly for readability: it is nicer to see a physically formatted hash that replicates the structure of the JSON document, and then convert that into a string, instead of having to wade through a 3000-character, one-line, string.

Please note that this is a full-blown integration test. However, it is based on `rack-test`, and not Rails' `ActionController::TestCase`. `Rack-test` is a very simple wrapper that fires real requests at a Rack app (which is our Rails application) and then provides the returned document and request headers.

It really doesn't do anything else, whereas `ActionController::TestCase` is implemented with more than 600 lines of code that try to guess many things, sometimes convert hashes into JSON, sometimes not, and so on. I've had so many bad experiences with the latter test concept that I completely banned it from my test suites.

Especially when it comes to testing consuming parts of our API you will see how trivial `rack-test` makes this task as it simply expects a document string as the request body that's being sent to the endpoint. Trying the same with Rails' built-in `TestCase` was a nightmare.

HAL Hypermedia Links

One thing you've probably noted is that the `_links` section in the document has a certain format. That's right! Instead of coming up with my own document syntax, I decided to use a well-defined

existing one.

[HAL](#)⁵⁴ does not only define how nested relation objects get rendered, it also has a format for embedding links into documents. This is usually something along the following.

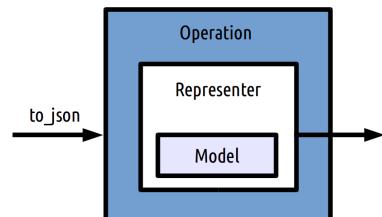
```
1  "_links": { ":rel": { href: ":path" } }
```

The special `_links` key marks the hypermedia section, links are keyed by their `rel`, which indicates their semantic or meaning, followed by various attributes for that link. In this book, we're only using the `href` attribute which contains the actual "clickable" URL or path of the link.

Also, what's the whole links thing about? Why am I putting links or URLs into that document? We'll discuss that more in a few minutes, but it is an unwritten law in the hypermedia community to always embed the URL that exposes the document into the document itself. This is called a *self link*.

Rendering

Now that we got a test in place, and we understand how and why the expected document looks the way it looks, how do we actually render that representation of the comment?



In Trailblazer, rendering happens via the operation. “*Hey, that’s too many concerns in one object, operation is a god class!*” you might find yourself yelling at me, now. Hang on, I didn’t say the rendering is implemented in the operation, I said rendering is orchestrated by the operation.

The operation that embraces logic for that particular HTTP endpoint simply keeps references to all objects it needs to render a document and instructs those to do so.

Let's go the other way round, ask yourself how would you manually implement the JSON serialization of a comment in Rails.

```
1  comment = Comment.find(params[:id])
2  json    = CommentSerializer.new(comment).to_json
```

You'd go grab the particular model instance and throw it into a serializer.

Now, let's convert that to Trailblazer. In order to do so, first, I add a `Show` operation in `app/concepts/api/v1/comment/operation.rb`.

⁵⁴http://stateless.co/hal_specification.html

```

1 module API::V1
2   module Comment
3     class Show < Trailblazer::Operation
4       include Model
5       model ::Comment, :find
6     end
7   end
8 end

```

By including `Model` and setting the `:find` action, the operation will take care of retrieving the correct model. If you wanted to, you could use this operation with your own serializer.

```

1 comment = API::V1::Comment::Show.present(params).model
2 json     = CommentSerializer.new(comment).to_json

```

Here, the operation fetches the record and exposes it via `model` (line 1). The external serializer renders the document (line 2). The latter could be from any framework, Yaks, ActiveModel::Serializer, you name it. This is absolutely legit. If you don't want to benefit from Trailblazer's representation framework, feel free to manually render documents.

Representer

Anyway, this chapter wouldn't have so many pages if that was it. Trailblazer comes with a very thin binding to Representable⁵⁵, a gem that provides what we call *representers*. A representer is a document declaration that allows rendering and parsing documents following the document specs.

Representers are usually classes with declarative definitions. Here's a quick example. Note that this code is absolutely not depending on Trailblazer.

```

1 class CommentRepresenter < Roar::Decorator
2   include Roar::JSON
3
4   property :body
5 end

```

Following the example from above, you could apply this representer to the operation's model.

⁵⁵The way I mix Roar and Representable might be confusing, but it's actually very simple. Representable is the gem that provides almost 95% of the representer mechanics. Roar extends that to a certain extend and adds support for hypermedia and media formats. In other words: Almost all features from Roar actually come from Representable, so bear with me and pardon the confusion.

```

1 comment = API::V1::Comment::Show.present(params).model
2 json     = CommentRepresenter.new(comment).to_json
3 #=> '{"body": "I love it!"}'

```

Again, the operation provides the model (line 1). After the decorator has been instantiated with the model, I call `to_json`. This call will invoke the representer's rendering and traverse its internal properties to compile a JSON document. In our simple example, this will end up as one read on the model where the representer invokes `model.body` and places that into the document.

Using Representer

Trailblazer allows to integrate representers directly into an operation. This works by including the `Representer` module that adds two methods.

```

1 module API::V1
2   module Comment
3     class Show < Trailblazer::Operation
4       include Model
5       model ::Comment, :find
6
7       include Trailblazer::Operation::Representer
8       representer do
9         property :body
10      end
11    end
12  end
13 end

```

The existing `Show` operation gets extended with `Representer` (line 7). This allows me to define representers inline using the `representer` method. As this will simply create an anonymous Roar`::Decorator` class behind the scenes, I can use the exact same API in the block (line 8-10).

We will soon learn that there's other ways to reference representers, but before talking about compositions, here's how you'd render this document.

```

1 API::V1::Comment::Show.present(params).to_json
2 #=> {"body": "I love it!"}

```

In fact, the `Representer` module also imports a `to_json` instance method into the operation for your convenience. Don't panic, this does absolutely not break any responsibility scopes, but simply orchestrates the representer and model.

Here's what happens inside the `to_json` method.

```

1 class Show < Trailblazer::Operation
2   # ...
3   def to_json(*)
4     self.class.representer.
5       new(model).
6       to_json
7   end

```

Now this is really convenient. The operation's `to_json` method does the decorating for you, as well as it dispatches the actual rendering. It grabs the representer class from the operation and instantiates the representer. The operation's `model` is being passed into the representer (line 4-5). The call to `to_json` will invoke the representer's rendering, just as we did it manually before (line 6).

To sum that up: an operation can take care of the JSON rendering if you want that. All you need to do is include `Representer`, define a representer and call `to_json` on the operation instance.

GET Controller

We now understand how operation, model and representer play together to serialize a JSON document. How are we gonna hook this into our API endpoint?

Well, in Rails, every journey starts with a route.

Here's what I add to `config/routes.rb`.

```

1 namespace :api do
2   namespace :v1 do
3     resources :comments
4   end
5 end

```

I'm not a Rails routing expert, however, the new nested configuration will, among others, give us the route `/api/v1/comments/1` which leads to `API::V1::CommentsController` and its `show` action.

The new controller sits in `app/controllers/api/v1/comments_controller`.

```

1 module API::V1
2   class CommentsController < ApplicationController
3     def show
4       render json: Comment::Show.present(params)
5     end
6   end
7 end

```

That is sufficient code to plug the operation with its representer to a controller and let it render JSON documents.

Step by step, what will happen now? When requesting the URL `/api/v1/comments/1` the above controller's `show` method will be run. In that action, I invoke the `Comment::Show` operation manually and pass the `params` hash into it.

Because of the way I arranged the namespace nesting, the `Comment::Show` here will be resolved to `API::V1::Comment::Show`, which is the operation with the representer we just wrote.

The operation will run its `present` code, find the model from `params` and return itself, completely analogue to what we did manually earlier.

Now, the tricky part. `render json:` thinks it is working with a model, and will invoke `to_json` on it. Anyway, the model here is the operation instance, but it still responds to `to_json` and returns the document which becomes the response body.

And here goes our first Trailblazer API endpoint!

Responders

Instead of having to manually invoke the rendering in the controller, Trailblazer provides the `respond` controller method. Note that you have to have the `responders` gem installed.

```

1 module API::V1
2   class CommentsController < ApplicationController
3     respond_to :json
4
5     def show
6       respond Comment::Show
7     end
8   end
9 end

```

Using the `respond` approach, the first steps are identical. The operation gets instantiated and invoked by passing in the controllers `params`. Then, the operation is passed to a responder. Here's what happens internally.

```

1 def respond(operation_class)
2   respond_with Comment::Show.(params)
3 end

```

The `respond_with` part is a bit hard to understand. Note that it receives the operation instance as first argument. The internal responder will now figure out a few things and query the operations about its `persisted?` state, and so on. Based on these information, the responder not only invokes `to_json` on the operation, but also sets `Location:` headers and more in the request.

In other words, the responder takes away quite some work to make that action comply with HTTP standards.

To allow the operation to communicate with the responder, you need to include the `Responder` module.

```

1 class Show < Trailblazer::Operation
2   include Model
3   model ::Comment, :find
4
5   include Responder
6   include Trailblazer::Operation::Representer
7
8   representer do
9     property :body
10    end
11 end

```

The inclusion will import some readers like `persisted?` for the responder (line 5).

Personally, I prefer the `respond` method as it unifies the way we invoke operations in API controllers and takes away the pain of returning the correct HTTP status code, content type, and other headers.

Inferred Representer

It might sound funny, but you now may run the test we wrote at the beginning of this chapter. And - it will break! That's because our representer and the expected document structure do not match. Before fixing this, I want to show two other ways how to specify representers.

A representer and a contract have much in common. They use the same declarative API, and even the same internal schema library. This allows to quickly infer a representer from a contract without having to write it yourself. To use this feature, the inferring operation needs a contract defined.

One way to provide a contract would be by inheritance.

```

1 class Show < ::Comment::Create
2   include Model
3   model ::Comment, :find
4
5   include Responder
6   include Trailblazer::Operation::Representer
7 end

```

Inheriting from `::Comment::Create`, the operation we implemented in earlier chapters, will copy its contract to the new `Show` operation. Since the `Representer` module is included, it will automatically provide an inferred representer as we do not provide one ourselves.

When rendering this operation, the following document will be generated from the inferred representer.

```

1 API::V1::Comment::Show.(id: 1).to_json
2 #=> '{"body": "Love it!", "weight": 1, "user": {"email": "fred@trb.to"} }'

```

As you can see by the included properties, the representer must have been roughly defined along the following lines.

```

1 representer do
2   property :body
3   property :weight
4   property :user do
5     property :email
6   end
7 end

```

And this is exactly the structure of the inherited contract from `Comment::Create`.

Representer inferal is a nice way to quickly get a stub endpoint set up. Usually, I end up writing an explicit representer, even though I have some redundancy in it. Also, most forms are a subset of the properties desired in a representer. Inferring is mostly helpful in an API-only application where you have a contract defined that maps to your document, and from that let Trailblazer build a representer automatically⁵⁶.

Composing Representer

The third alternative, and this is the way we will go, is to keep representers in separate files and reference them from the operation. By maintaining external representers, we minimize coupling

⁵⁶Traiblazer gives you a rich set of functions to infer, inherit and refine representers. Please refer to [the docs](#) to learn more about this.

between the components and I simply find it easier to navigate since I instantly know where to look for representers.

Per concept, I keep all representers in one file. As always, it is totally up to you whether or not you like one file per class, but as we will soon see, sometimes a representer is nothing but a few lines of code, so for me it's fine to group them in one physical file.

For the API::V1::Comment concept, representers go into `app/concepts/api/v1/comment/representer.rb`. And, to finally get our test green, here's the content of that file.

```

1 module API::V1
2   module Comment
3     module Representer
4       class Show < Roar::Decorator
5         include Roar::JSON::HAL
6
7         property :body
8
9         link(:self) { api_v1_comment_path(represented.id) }
10      end
11    end
12  end
13 end

```

The main asset here is the `Show` class, derived from `Roar::Decorator` (line 4). This is what happens behind the scenes when you use the `representer` block in your operation class. I namespace this representer in the `Representer` namespace the way we did it with contracts, too (line 3).

The included `Roar::JSON::HAL` module imports basic JSON semantics, plus support for hypermedia (line 5). Apparently, we can add hypermedia links with the `link` class method. In this block, we can use Rails URL helpers to specify the `href` value of the rendered link. Where you'd normally pass the model's id into the path helper, we use `represented.id`, which will reference the actual `Comment` model's id that the representer decorates (line 9)⁵⁷.

Now that we have the representer in place, all that's missing is reference that class in the operation.

⁵⁷Currently, you need to manually mix in Rails' URL helpers to use them. While this will most probably be done automatically in `trailblazer-rails`, you can check out how I do it [here](#).

```

1 module API::V1
2   module Comment
3     class Show < Trailblazer::Operation
4       # ...
5       representer Representer::Show
6     end
7   end
8 end

```

Plugging the external representer into the Show operation is no more than passing the constant to representer (line 5). Note how I have to reference the constant Representer::Show, only. Ruby will look in the API::V1::Comment namespace automatically.

Running the integration test will result in a nice, a bit too flashy, but motivating green output on the console. We've everything implemented to present a comment as a JSON document via HTTP.

Rendering Nested Documents

Before we move on to the fun part of implementing document APIs, which is definitely the parsing, let's finish up the comment document. It should also show the comment author as a nested document fragment.

I extend the first test case for this.

```

1 describe "GET /comments/1" do
2   # ...
3   it "renders" do
4     get "/api/v1/comments/#{comment.id}"
5
6     last_response.body.must_equal(
7       {
8         body:   "Love it!",
9         _embedded: {
10           user: {
11             email:  "fred@trb.to",
12             _links: { self: { href: "/api/v1/users/#{comment.user.id}" } }
13           }
14         },
15         _links: { self: { href: "/api/v1/comments/#{comment.id}" } }
16       }.to_json
17     )
18   end

```

The comment document now contains the `user` as a nested fragment (line 10-13). That user fragment itself has a `_links` property that points to the user's path (line 12). In HAL, nested objects can contain links, too, helping the client browsing to nested resources without having to guess the URL.

We also have an `_embedded` key nesting the user in the document into a separate section (line 9-14). Another HAL semantic I've added to the document without telling you. HAL specifies a top-level object, which is the comment's properties, but it allows nested, associated objects. Those are listed under the `_embedded` key.

Don't confuse that with associations. While in most cases, the embedded fragments represent models that are associated with the top level model, this could also be arbitrary data that might be needed on the client side.

Anyway, our test now fails, 3 minutes after we fixed it eventually.

The fix is simple and happens in `app/concepts/api/v1/comment/representer.rb`. Here is the updated code for the comment API representer.

```

1 class Show < Roar::Decorator
2   feature Roar::JSON::HAL
3
4   property :body
5   property :user, embedded: true do
6     property :email
7     link(:self) { api_v1_user_path(represented.id) }
8   end
9
10  link(:self) { api_v1_comment_path(represented.id) }
11 end

```

Unmissingly, I've added the nested `user` property and its `email` property (line 5-8). The nested `link` renders the user link (line 7). Here, I use a different URL helper. In this context, `represented` will refer to the nested user model.

We can configure the HAL mechanics to treat the nested user as an "embedded" property by adding `embedded: true` to the property declaration (line 5).

To make this work, I had to change `include Roar::JSON::HAL` to `feature HAL` (line 2). Using the `feature` to include a module will automatically include this particular module into nested representers. That way, I can use `link` in the user representer, too.

Since we use the `api_v1_user_path` helper, a resource has to be added to `config/routes.rb`.

```

1 namespace :api do
2   namespace :v1 do
3     resources :comments
4     resources :users

```

Tests are green again after this change, and after cheering for a few seconds we can finally move on to something more exciting.

Scopes and Sorting

An interesting requirement would be to implement an index operation that lists things, as in GET /api/v1/things. That alone is a quite straight-forward task, let's spice it up by allowing different data sets. Those data sets will be instructed using query parameters like things?sort=recent and sort the things as specified.

A test speaks more than a thousand words. What I do is I add another describe block to test/api/v1/things_test.rb.

```

1 class ApiV1ThingsTest < MiniTest::Spec
2   describe "GET /things" do
3     let (:things) do
4       20.times.collect { |i|
5         Thing::Create.(thing: { name: "Thing #{i}",
6           users: [{"email"=> "#{i}@trb.to"}]}).model
7       end
8     end

```

I first setup 20 random things with names “Thing 1”, etc. Each thing has an author with a similarly structured email address (line 3-8).

```

1 describe "GET /things" do
2   # ...
3   it "sort=oldest" do
4     get "/api/v1/things?sort=oldest"
5
6     JSON[last_response.body].must_equal(
7       {
8         "_embedded"=>
9           {
10             "things"=>
11               things[0..8].collect do |t|

```

```

12      { "name"=> "#{t.name}" ,
13      "_links"=>{ "self"=>{ "href"=>"/api/v1/things/#{t.id}"}}}
14    end
15  },
16  "_links"=>{ "self"=>{ "href"=>" /api/v1/things?sort=oldest" } }
17 })
18 end
19 end

```

This test case sends a GET request to the `/api/v1/things` route, but adds the `sorted=oldest` query parameter that we're going to process in the `Index` operation (line 4).

The remaining test asserts in a very awkward way that correct items are embedded in the document (line 6-17). Also, note the `self` link that refers to this resource itself, includes the dynamic request parameter `sort`.

To allow this new GET route, I add a few lines to `config/routes.rb`.

```

1 namespace :api do
2   namespace :v1 do
3     resources :things

```

The `things` resource will now route requests to `/api/v1/things/` to the `index` action of the `API::V1::ThingsController`, which gets extended as follows.

```

1 module API::V1
2   class ThingsController < ApplicationController
3     def index
4       respond Thing::Index, present: true
5     end

```

Fairly simple, we just delegate the main work to `API::V1::Thing::Index`, which will be implemented in `app/concepts/api/v1/thing/operation.rb`.

```

1 module API::V1
2   module Thing
3     class Index < Trailblazer::Operation
4       include Trailblazer::Operation::Representer, Responder
5       representer Representer::Index
6
7       def model!(params)
8         return ::Thing.oldest if params[:sort] == "oldest"
9         ::Thing.latest
10      end
11
12      def to_json(*)
13        options = { user_options: {} }
14        options[user_options][:params] = @params
15
16        super(options)
17      end
18    end

```

A plain new operation class will implement the index logic (line 3). Including `Representer` and `Responder` allow to specify a representer and integrate `to_json` with the representer and model (line 4).

Using `representer`, I reference the external `Index` representer that we still have to implement.

Index Operation

Since this is a collection operation, we can't use the `Model` module to help us finding the correct rows. We could, but it would be more pain than doing it manually.

To find models with your own logic, you can simply override `Operation#model!` which is designed for just that.

I access the methods `params`, which are the arguments being passed into the operation call, and find out what the `:sort` value looks like (line 7-10). Depending on the value, I dispatch to either the `oldest` or `latest` scope of the `Thing` model. While I could also introduce a query object or construct queries in the operation, I find ActiveRecord's scopes pretty handy.

Note that I do need to address the model class using the root notation `::Thing`, otherwise Ruby will resolve this to `API::V1::Thing` which is a namespace and not a model.

Also, in order to access the sorting parameters in the representer to render a proper `self` link, we need to pass in the parameters into the `to_json` call when invoking the representer instance. This is why I override `to_json` in the operation (line 12-17). I create a nested hash that looks like the following snippet.

```

1  {
2    user_options: {
3      params: @params
4    }
5 }
```

This is Representable's API to inject dynamic options into render and parse calls. Any non-library options go into the `:user_options` value, and I pass the entire params under the same-named key. I access the operation's params using the instance variable `@params`.

I then call `super(options)` to invoke the operation's original rendering with my own options (line 16). In pseudo code, the rendering invocation of the operation will resolve to the following snippet being executed.

```
1 Index.representer.new(model).to_json(user_options: { params: @params })
```

As a next step, we need to implement the search logic in the model class by adding the `oldest` scope. That happens in `app/models/thing.rb`.

```

1 class Thing < ActiveRecord::Base
2   scope :latest,  lambda { all.limit(9).order("id DESC") }
3   scope :oldest,  lambda { all.limit(9).order("id ASC") }
```

This is just another scope on the model, and as ActiveRecord does a great job at that, it's not incorrect to put that into the model class (line 3).

Quite important is to understand that `model!` is the place to combine dynamic request parameters with query logic. This is where pagination, scoping, sorting, and filtering happens. We will see more of that in the next example, again.

To round up this implementation step, we need to implement the representer in `app/concepts/api/v1/thing/representer.rb`.

```

1 module API::V1
2   module Thing
3     module Representer
4       class Index < Roar::Decorator
5         feature Roar::JSON::HAL
6
7         collection :to_a, as: :things, embedded: true, decorator: Create
8
9         link(:self) do |params:, **|
10           options = {}
11           options[:sort] = params[:sort] if params[:sort]
12
13           api_v1_things_path(options)
14         end
15       end

```

The `Index` representer is derived from a plain `Roar::Decorator`, but I include the `HAL` module to make it compliant to our desired media format (line 4-5).

I then define a collection property `to_a`, that I alias to `:things`, mark it as an embedded fragment, and provide an explicit representer instead of using an inline representer (line 7). Don't you worry, we'll discuss this odd piece of code in a second.

The last block of this representer defines the `self` link for the rendered document (line 9-14). Blocks for `link` can contain arbitrary code and receive an optional hash. This hash passed into every `link` block is the `:user_options` argument we gave into the top-level `to_json`.

By using keyword arguments I let Ruby filter out the `params` value (line 9). I then create a hash `options` that I pass into the URL helper (line 10-13). In this hash, I add `:sort` so the URL helper will include our dynamic request option into the rendered document.

It is pretty common in the Roar world to have more than just a call to a URL helper in a `link` block. What we do here could be nicely abstracted, but as always, this book shows the explicit way for a better understanding.

Lonely Collections

Let's talk a bit more about the representer property.

```
1 collection :to_a, as: :things, embedded: true, decorator: Create
```

To understand what is going on here we need to take a step back and have a look at how we use the representer. The `Index` operation will fetch a collection of `Thing` models and pass this array to the representer.

If you were to use the normal `Thing::Create` representer to render the collection, `Representable` would try to call property getters on the array.

```
1 ary = Thing.latest #=> [<Thing>, <Thing>]
2 ary.name #=> Exception!
```

It is quite obvious that a singular representer can't be applied to a collection as it will treat the array as the "model" itself, and this won't work.

`Representable` comes with built-in support to generate a collection representer for you on-the-fly. However, this would literally render an array of things. The structure we require is bit more complicated as we want a real HAL document with exactly one embedded property, which is our collection of items.

```
1 {
2   "_embedded"=>
3   {
4     "things"=> []
5   }
6 }
```

According to my friend Mike Kelly, the creator of HAL, this is best practice for representing collections in HAL.

Achieving this structure is relatively simple by using a `collection` property and mark it as `embedded: true`. So far, so good. Since we pass an array into the representer, `Representable` will now call `ary.things` to retrieve that collection. To avoid a `MethodMissing` exception, I simply call the property `:to_a` and alias it to `:things`.

When rendering the document for an array of things, the representer will call `ary.to_a`, which will return the array itself, and then render it with the key `:things`. Exactly what we wanted.

The actual items need a representer, too. While I could use an inline representer, I reference the existing one using `decorator: Create`. Using the `:decorator` option is another technique to compose representers without creating redundancy.

Anyway, running the test suite will now pass, our sorted collection is serialized exactly the way we need it, and even contains dynamic parameters in the self link. Pretty cool.

Filtering

It actually wasn't too hard to implement sorting, was it? Once you grasp that the operation is responsible for collecting, and it's the representer's job to serialize that object graph, the layering starts making sense.

As a last requirement, I want to allow API users to skip rendering of specific fragments of the index document. As per now, that document contains things, which in turn are composed of author and comment fragments.

Assuming that Gemgem gains a lot of traction, and many comments might be embedded per thing, it might be a good idea to allow turning off rendering for comments and/or authors.

I will skip huge parts of the test code here as they are big chunks of JSON, and only some parts are relevant for understanding. Speaking of, here's the next test I put into `test/api/things_test.rb`.

```

1 class ApiV1ThingsTest < MiniTest::Spec
2   describe "GET /things" do
3     it "includes comments, only" do
4       dhhs_thing = Thing::Create.(thing:
5         {name: "Rails", users: [{email=> "dhh@trb.to"}]}) .model
6       comment      = Comment::Create.(id: dhhs_thing.id, comment:
7         {body: "I like his stuff!", weight: "1",
8          user: {email: "jose@trb.to"}}) .model
9       # ...

```

I first create a massive fixture with several things from different authors (line 4-9). To some, I add comments so we can test the rendering (line 6-8).

```

1 it "includes comments, only" do
2   # ...
3   get "/api/v1/things?include=comments"
4
5   JSON[last_response.body].must_equal (
6     {"_embedded"=>
7      {"things"=>
8        [{"name"=>"Rails",
9         "_embedded"=>
10        {"comments"=>
11          [{"body"=>"I like his stuff!", "weight"=>1,
12           "_embedded"=>{"user"=>{"email"=>"jose@trb.to",
13             "_links"=>{"self"=>{"href"=>
14               "/api/v1/users/#{comment.user.id}"}}},
15           {"_links"=>{"self"=>{"href"=>
16             "/api/v1/comments/#{comment.id}"}}},
17           {"_links"=>{"self"=>{"href"=>"/api/v1/things/#{dhhs_thing.id}"}}},
18           # ...
19           {"_links"=>{"self"=>{"href"=>"/api/v1/things"}}}
20         }

```

```
21      )
22 end
```

Using `GET /api/v1/things?include=comments` I retrieve the document that must only contain the thing's attributes plus nested comments, and no author details (line 3).

In a convoluted matching test I make sure that no author fragments and only comments are to be found in that document (line 5-18). Again, I spare you the details of the hashes here, as they're accessible in the repository.

To make this example as brief as possible, I also left out including query parameters in the self link here (line 20).

`Representable` allows to instruct inclusion and exclusion of properties. As an additional service, you can also nest this, and thus fine-tune the inclusion of fragments. For example, when rendering a thing, here's how you could include the nested comments' bodies, only, and exclude the weight and author.

```
1 ThingRepresenter.new(thing).to_json(comments: {include: [:body]})
```

A really handy mechanic to provide filtered documents for nested data structures.

Implementing this feature doesn't touch the representer directly. Instead, we use the `:include` option of `Representable` that we just discussed. The filtering, from our perspective, takes place in the `Index` operation.

```
1 module API::V1
2   module Thing
3     class Index < Trailblazer::Operation
4       #
5       def to_json(*)
6         #
7         if @params[:include]
8           scalars = self.class.representer.definitions.get(:to_a).
9             representer_module.definitions.values.
10            reject { |dfn| dfn.typed? }.
11            map { |dfn| dfn[:name].to_sym }
12
13         options[:to_a][:include] = [
14           *scalars, :links, @params[:include].to_sym]
15       end
16
17       super(options)
18     end
```

In the overridden `to_json` method, I add including options (line 13-14). The `options` hash will look like the following fragment in our test.

```
1 { to_a: { include: [:name, :links, :comments] } }
```

Since it's passed to the original `to_json` method, `Representable` will take care of filtering for us (line 17).

Representable's Internal API

A bit of manual work is necessary to include the instructed nested comments, and the original scalar values. Even though this code is a bit ugly, I decided it's a good chance to teach you some more about `Representable`'s internal public API.

To make the representer include the nested comments along with the scalar properties of the thing - which is `name`, only, in this example - we need to find out those scalars at runtime. We could simply provide them hard-wired, but going the complicated, dynamic way is way more fun.

```
1 scalars = self.class.representer.
2   definitions.get(:to_a).
3   representer_module.definitions.values.
4   reject { |dfn| dfn.typed? }.
5   map { |dfn| dfn[:name].to_sym }
```

I can access the operation's representer using `self.class.representer` (line 1). There's only one property named `to_a`, which I can grab using `definitions.get` (line 2). This is an instance of a `Definition` and contains the nested representer (where we define the thing's actual properties, like `name` and `link`) via `representer_module` (line 3).

This, again, is just another representer. Actually, that's `API::V1::Thing::Representer::Create`. I access all of its property definitions with `definitions.values` (line 3). To filter out nested properties I use `reject` and test for nestedness using `typed?` (line 4).

The result is a set of definitions that are not nested. I read out the names and symbolize it (line 5).

The `scalars` variable will now be a one-element array that looks like `[:name]`. A lot of work for a little array.

`Representable` has that built in, but, you must admit, this was interesting!

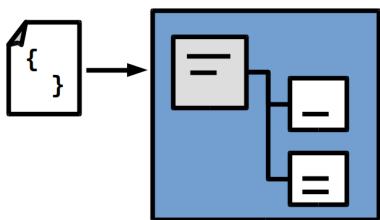
When running the test, the operation will find out if a request parameter instructs filtering, compute the set of properties to render, pass that to the representer using the `:include` option and the resulting document is exactly what we were expecting.

We know everything about rendering now, and can move on to parsing, which I find more fascinating.

Hypermedia APIs: Deserialization

After having discussed the essentials of rendering documents using representers, we should focus on a much harder part in software development: parsing, or deserialization, of incoming documents.

Deserialization means to parse a document into fragments, and then assign those fragments to an object graph. This object graph, as an example, could be a nested model. The fragments could be attributes.



Deserializing a document to an object graph.

In Rails, there simply is no deserialization. In a PUT, PATCH or POST request, “deserialization” is reduced to a parsed, deeply nested params hash that is either thrown into an unhappy update_attributes hoping that all will go well. Or it is a messy manual traversal of the params hash where many ifs and elses will try to set up an object graph that is close to the models nested in the incoming representation.

While this might look nice with one level of simple attributes, the madness quickly begins when it comes to creating or retrieving nested objects from the document, adding, replacing or deleting particular items in collections, or invoking special finders or setters on nested objects.

It really gets amusing when you have to whitelist parameters, too, since `strong_parameters` usually breaks down here.

The real fun starts when you need to add items to collections but need to validate that new collection before persisting it. For example, when the restriction is “*Only 3 items maximum in this collection!*” things get really hairy in Rails when “deserializing”.

Every project I’ve worked on so far solved this in a different way. I’ve seen attempts to utilize `accepts_nested_attributes_for`, manual traversals using the Hashie gem or other half-baked approaches that simply do not hide the fact that deserialization is completely ignored in Rails (and almost all “REST” gems).

Not only does the complete lack of a crucial layer defeat the purpose of a framework, it also creates redundant code: Knowledge about structure and semantics of your document is now hard-wired into serializers or Rabl templates, and into the parsing code. If you change anything, you have to fix it in both places, and at some point will forget to do so.

What a horrible picture am I painting here? Let’s forget about all that. In Trailblazer, deserialization is a first-class citizen of our workflow and we’ve thought many years about how to do this neatly, and have it working in many production apps backed by the Representable gem.

Deserialization in Trailblazer

You surely remember our exemplary mini representer from earlier, the one that could render a comment document.

```

1 class CommentRepresenter < Roar::Decorator
2   include Roar::JSON
3
4   property :body
5 end

```

This class is called *representer* because it *represents* an object graph. That means, it can render and parse. Internally, two completely separated components will take care of those two tasks, but they will both use the same representer definition as the document structure, so they know which properties to render or parse, where to apply nesting logic, and so on.

You already know how to use the representer for rendering, therefore, let's discuss the parsing aspect.

```

1 comment = Comment.new
2 CommentRepresenter.new(comment).
3   from_json('{"body": "I love it!"}')
4
5 comment.body # "I love it!"

```

Parsing⁵⁸ with representers is super simple. The representer will grab all fragments it knows from the document and assign them to the decorated object by calling a setter method.

In that example, this will be a `comment.body = "I love it!"` - nothing more. The document could be loaded with other bogus properties and fragments, the representer will simply ignore those, giving you a whitelisting mechanism for free.

This sounds pretty familiar, doesn't it? It feels like Reform. And that's because Reform internally uses a representer for deserializtion in `validate`. We'll speak about this soon enough.

Before discussing how parsing, the form object and the operation play together, let me quickly show you how a nested parsing would take place.

What if the JSON document looked like this?

```
1 nested_json = '{"body": "I love it!", user: {"email": "fred@trb.to"}}'
```

There now is a scalar property and a nested user object in the document. Representers can have nested representers, and once you tell Representable what object type they are, it will be able to parse a nested document.

⁵⁸I will say *parsing* instead of *deserializing* every now and then, even if it's not the exact same thing.

```

1 class CommentRepresenter < Roar::Decorator
2   include Roar::JSON
3
4   property :body
5   property :user, class: User do
6     property :email
7   end
8 end

```

I simply nest another representer into the existing one (line 5-7). The `:class` option will help Representable parse the nested fragment into a `User` object. This is the most primitive parsing mechanism, but it helps understanding its concepts.

```

1 comment = Comment.new
2
3 CommentRepresenter.new(comment).
4   from_json(nested_json)
5
6 comment.user #=> <User email: "fred@trb.to">
7 comment.user.email #=> "fred@trb.to"

```

When parsing, the nested `user` fragment is “turned into” a `User` object, and the nested properties are deserialized onto that newly created model.

Again, this is a very primitive mechanic, Representable can do rich population, find models by arbitrary fields instead of creating them, and so on. Since Reform uses representers for deserialization, the semantics here are almost identical⁵⁹.

Parsing Comment

We’re about to implement the deserialization of a comment with a nested user, similar to the comment form we created in the earlier chapters, the one that sits on the thing’s show page.

As a first, yet triumphant, step, I add another test to `test/api/comments_test.rb`. Again, it all goes into its own `describe` block.

⁵⁹I won’t discuss populators in Representable and Roar in this book, as population in Trailblazer happens via the form. However, should you need separate mechanics, you can learn everything you need to know in the [official, free documentation](#).

```
1 describe "POST /api/v1/things/1/comments" do
2   let (:thing) { Thing::Create.(thing: {name: "Rails"}).model }
3   let (:json)  { {
4     body: "Love it!", weight: "1",
5     user: { email: "fred@trb.to" }
6   }.to_json }
```

First, I create the thing we're going to comment (line 2). In the second let I define the POST document body that will be pushed to the endpoint (line 3-6). As you can see, I nest a user into the document. Please note that right now, I'm *not* HAL-conform since the document is missing the _embedded level. I chose to go that way for the sake of clarity. We'll correct the wrong input document format in a minute.

```
1 describe "POST /api/v1/things/1/comments" do
2   # ...
3   it do
4     post "/api/v1/things/#{thing.id}/comments", json
5
6     comment = thing.comments[0]
7     last_response.status.must_equal 201
8     last_response.headers["Location"].
9       must_equal "http://example.org/api/v1/comments/#{comment.id}"
```

Once the environment is set up, a post will invoke the request we want to test (line 4). See how the things/:id is prefixing the actual comments endpoint? I use a nested route here so I don't have to pass the thing ID as a parameter. Remember, in order to create a comment, we need `thing_id` and Rails helps us with that by providing nested routes.

I know, you're impressed by my testing discipline. But be honest, it is so much simpler to write this little, automatic test than having to fumble around with Postman or Curl.

After the request went through, I grab the comment that we hopefully created (line 6).

The next two tests make sure the response was a 201: Created HTTP status code (line 7). The "RESTfulness" of my response is tested when I assert that the `Location:` header points to newly created resource. This is another best practice in the hypermedia world.

In order to verify that the POST worked, I add two simple assertions for now.

```

1 describe "POST /api/v1/things/1/comments" do
2   it do
3     # ...
4     comment.body.must_equal "Love it!"
5     comment.user.email.must_equal "fred@trb.to"
6   end

```

The test makes sure the correct data was persisted and the right nested objects got created (line 4-5).

Alternatively, you could issue another GET request to the new URL and test the response document. I added an example in the repository for that.

Making this test pass is surprisingly simple, thanks to some nice features in Trailblazer.

As always, we need to define the new POST route in config/routes.rb.

```

1 namespace :api do
2   namespace :v1 do
3     resources :things do
4       resources :comments
5     end

```

I added a nested resource (line 3-5) which will route requests to /api/v1/things/:id/comments to Api::V1::CommentsController. This is one of Rails' features that I actually do appreciate.

The API comments controller is extended with a create action to intercept POST requests.

```

1 module API::V1
2   class CommentsController < ApplicationController
3     # ...
4     def create
5       respond Comment::Create,
6       namespace: [:api, :v1]
7     end

```

To run the operation, I use the familiar respond and reference the API::V1::Comment::Create operation (line 4). The mysterious :namespace option is directly passed to Rails' responder when the response is compiled and sent. The responder includes a Location: header in the response and needs some fine-tuning to find the correct URL helper.

Next, I add the Create operation to app/concepts/api/v1/comment/operation.rb.

```

1 module API::V1
2   module Comment
3     class Create < ::Comment::Create
4       include Trailblazer::Operation::Representer, Responder
5     end

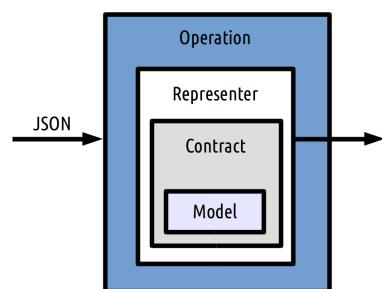
```

Note that I inherit from `::Comment::Create`, the original form processing operation from earlier chapters (line 3). I include the `Representer` and `Responder` module to make the operation work with the responder, as discussed earlier (line 4).

And, believe it or not, but this is enough code to make the test run.

Deserialization and Contract

There's two things to note here. First, we send in the wrong document, it doesn't follow the HAL specifications and is missing the `_embedded` key.



Second, since we inherit from another operation, the contract is inherited. As we use `Representer`, a representer will automatically be inferred, allowing us to deserialize a JSON document, validate it using the form, process and persist it using the inherited operation mechanics.

Now, why on earth does this work without us having to do anything?

The contract that we inherited from the `Comment::Create` operation doesn't really care whether its input comes from a form submission, from the command line, or a JSON document!

Here's what happens from top to bottom.

1. We fire a POST request at the `create` action. Its body is a JSON document.
2. The controller action parses that document into a `params` hash. It then dispatches to the API `Create` operation and passes in this very `params` hash.
3. In the operation, the inherited `process` method is called and in it, the `validate` method will receive this hash. The operation knows that it needs a special representer since we mixed in `Representer` and instantiates it.
4. Now, the representer is called to deserialize this hash, which means it traverses it and assigns known fragments to known properties.
5. Remember, the structure of the JSON document is “accidentally” identical to the form hash that we normally have to deal with. That means the representer that was automatically inferred from the contract “understands” this hash and knows what properties to deserialize.

6. The crucial point to understand this whole cycle is: the representer deserializes the hash *to the contract*. In other words, it goes through the hash, grabs the body fragment and writes it to the contract using a setter as in `contract.body = "Love it!"`. The contract doesn't even know its being reused for a JSON API, since it doesn't see the incoming document.
7. When inferring the representer from the contract, populator options are considered, too. When the representer reaches the nested user fragment, it reaches out to the contract, invokes its populator, and continues the nested deserialization. All private population logic happens on the form, not in the representer.
8. After the contract is populated with all the attributes and nested instances, the normal process flow will continue, unaware that it is validating and persisting a JSON document request.

By separating representer and contract, we can reuse the same form with different input sources. While the representer knows about the incoming document structure and traversal rules, the form provides setters for properties and will handle the population for nested objects.

Deserialization in Reform

I know this was a huge chunk of information. Let me briefly explain to you how deserialization and validation in a normal Reform form work, in a much simpler environment, cutting out all the Traiblazer operation noise. This will help understanding that we simply replace the representer in a JSON environment.

Normally, when you invoke a Reform form, you call its `validate` method and pass in a hash that matches its property structure.

```

1 form.validate(
2   body: "Love it!",
3   user: {
4     email: "fred@trb.to"
5   }
6 )

```

Internally, Reform will now assign all scalar values via its setters.

```

1 form.body = "Love it!"

```

Nested fragments work slightly different. First, Reform will invoke the populator to create the nested form. In our case this will be a nested user form that itself wraps a `User` model.

It then simply passes the nested hash to the nested form's `validate` method and, recursively, the cycle starts again.

In pseudo code, this roughly looks as follows.

```

1 class Reform::Form
2   def validate(params)
3     self.body = params[:body]
4     self.user = run_populator_for(:user)
5     self.user.validate(params[:user]) # recursion.
6
7     # run actual validation.
8   end
9 end

```

The difference between what I just illustrated⁶⁰ and the real code is: Reform internally *always* uses a representer to deserialize the hash.

So, a bit closer to reality, this happens.

```

1 def validate(params)
2   internal_representer.new(self). # self is the form instance itself.
3   from_hash(params)
4
5   # run actual validation.
6 end

```

Regardless of all the details that happen here, Reform uses a genuine representer to do the deserialization, even in a non-JSON environment. You've used this mechanism many times before without knowing it.

The internal representer will do exactly the same I did by hand in the first pseudo code snippet. It assigns scalar values to the form, and calls the contract's populator code which returns the nested form. It then recurses on that nested object.

The validation after that is completely unaware of the former deserialization. Validation checks the form's state and decides about validity.

Likewise, deserialization is completely decoupled from validation.

Now, why am I explaining this? The reason we suddenly can read, validate and process documents in the HAL format is because we swap the form's internal representer. This especially makes sense when the document format and the form's structure diverge - and that is the case when we use "real" HAL for our POST request.

Deserializing HAL

You're still with me, and that's great. As a next step, I want to POST a valid HAL document to the Comment::Create operation. I'm back in the test/api/comments_test.rb file.

⁶⁰The upper part of that method actually is implemented in Form#deserialize.

```

1 describe "POST /api/v1/things/1/comments" do
2   let (:json) do
3     {
4       body:      "Love it!", weight: "1",
5       _embedded: { user: { email: "fred@trb.to" } }
6     }.to_json
7   end

```

All I do in this file is change the json fixture slightly. Instead of pushing the `user` : key as a top-level property, I nest it in under the `_embedded` key the way HAL specifies it (line 5).

The test will now break, as we send a HAL document. Our operation still uses an inferred representer that has no clue about HAL. It simply doesn't "see" the `_embedded` fragment, hence it does not add a user, the form's validation fails, the operation will be invalid, customers will be angry and the earth will collapse.

To escape this dark apocalypse, we should use a representer that knows HAL in the `API::V1::Comment::Create` operation.

The API's Create operation gets slightly enhanced.

```

1 class Create < ::Comment::Create
2   include Trailblazer::Operation::Representer, Responder
3
4   representer Representer::Show
5 end

```

I simply reference the representer from the Show operation which is a valid HAL representer. Remember, we added the `embedded: true` options, hyperlinks and all that in `Representer::Show` earlier.

Now, there's one tiny more thing I have to do in this representer.

I open `app/concepts/api/v1/comment/representer.rb` and add one line.

```

1 class Show < Roar::Decorator
2   feature Roar::JSON::HAL
3
4   property :user, embedded: true,
5   populator: Reform::Form::Populator::External.new do
6     property :email
7     link(:self) { api_v1_user_path(represented.id) }
8 end

```

The representer is exactly as it was before except for the `:populator` option I added to the `user` property (line 5).

When running the tests this time, everything will pass. We parsed an incoming HAL document with a specialized representer, deserialized the data to the unknowing form, and persisted the changes reusing operation logic we wrote chapters ago.

To summarize this, again, here's an even quicker run-through. This time, I'll skip the request to controller to operation steps.

1. The HAL hash is passed into `API::V1::Comment::Create`'s `#process` method. This method is inherited from `Comment::Create`, and it will further pass this HAL hash into `validate`.
2. The operation has the Representer module included and therefore is expecting HAL. It instantiates the HAL representer `API::V1::Show::Representer` that we reference in the operation class.
3. This representer will now deserialize the hash with all its HAL internals to the contract instance. The contract is inherited from the generic form operation and does not know a thing about HAL. This doesn't matter, as the representer takes care of this.
4. When the representer encounters a nested fragment, it calls the respective populator. I defined `populator: Reform::Form::Populator::External.new`, which means the representer will call the form's populator. We don't have to copy the complicated logic.
5. Once deserialization is finished, the contract validates its virtual state.
6. If found valid, the contract persists data via the model. This is all happening in logic inherited from the generic operation.

Representers are extremely powerful. They are an explicit document description and can render and parse documents without distributing knowledge about document structure and semantics across the entire framework.

They are highly reusable, which we could see the way `Show::Representer` is used for rendering, but also to deserialize incoming HAL documents. They can be used to transform complicated hypermedia documents to Trailblazer contracts or stand-alone twins.

Most of the processing logic could be inherited, we didn't have to write a single line of code. If requirements should start to diverge for different environments, you can easily override parts or the entire process step.

Create

In the following sections I want to show you how we can apply our polymorphic operation logic of the `Thing` concept to API code. You know, where we expose different behavior for anonymous,

signed in and admins. This involves allowing users to authenticate. In Gemgem, I use Basic Auth straight from HTTP, but you're free to implement it with any kind of authentication you like.

Let's start to implement POST request processing for the endpoint /api/v1/things.⁶¹ As you might recall, creating things worked without signing in, so we won't see too much polymorphic behavior here.

Test cases for the things API will go into test/api/things_test.rb.

```

1 class ApiV1ThingsTest < MiniTest::Spec
2   describe "POST" do
3     it "allows adding authors" do
4       json = {
5         name:      "Lotus",
6         _embedded: { authors: [{ email: "fred@trb.org" }] }
7       }.to_json
8
9       post "/api/v1/things/", json

```

I setup a sample JSON document to POST (line 4-7). Note that this is perfectly formatted HAL, already, so we have to make sure the processing representer is prepared for that.

The document then gets sent to the endpoint (line 9).

```

1 it "allows adding authors" do
2 # ...
3   id = Thing.last.id
4   author_id = Thing.last.users.first.id
5
6   last_response.headers["Location"].
7     must_equal "http://example.org/api/v1/things/#{id}"
8   assert last_response.created?

```

In the second part of the test I grab the created Thing's ID as well as the author's ID (line 3-4).

Next, the Location: header is tested (line 6-7). I also make sure the returned HTTP status is 201. This can be tested using created?, too (line 8).

⁶¹I've left out discussing implementation of /api/v1/things/1 and the API::V1::Thing::Show operation, as it is trivial and analogue to what we've done here. You can find the code and tests anyway, in the Gemgem repository.

```

1 it "allows adding authors" do
2   # ...
3   response_json = {
4     "name": "Lotus", "_embedded": {
5       "authors": [
6         {"email": "fred@trb.org",
7          "_links": {"self": {"href": "/api/v1/users/#{author_id}"}}
8       ],
9       "_links": {"self": {"href": "/api/v1/things/#{id}"}}
10    }.to_json
11
12   last_response.body.must_equal response_json
13 end

```

Rails' responders per default return the rendered model in a POST request. In our case, the responder will run the operation and then call `to_json` on it, exactly the way it works in GET requests, too.

This is the perfect chance to test if the creation worked. This time, instead of asserting database state, I simply test the returned document and make sure IDs are assigned and the author is nested (line 12).

Note that the document says `authors` and not `users`. I renamed that fragment on purpose, and you will learn how in the next section.

The test for creating a `Thing` is hereby completed.

Now, don't get confused, but this time we do not need to add routes. This has been done earlier in this chapter and requests to `/api/v1/things` get safely routed to the controller that resides in `app/controllers/api/v1/things_controller.rb`.

```

1 module API::V1
2   class ThingsController < ApplicationController
3     respond_to :json
4
5     def create
6       respond Thing::Create, namespace: [:api, :v1]
7     end

```

This is the same conceptual implementation we had in our `CommentsController` earlier. I use `respond`, reference the `Thing::Create` operation and provide hints for the `Location:` header for the responder (line 6).

The processing operation, `API::V1::Thing::Create`, sits in `app/concepts/api/v1/thing/operation.rb`.

```

1 module API::V1
2   module Thing
3     class Create < ::Thing::Create
4       include Trailblazer::Operation::Representer, Responder
5
6       representer Representer::Create
7     end

```

Again, I inherit from the API operation's counter-part, the "real" operation `Thing::Create` (line 3). While the inheritance gives me the contract and the remaining behavior of the original `Create`, I reference an external representer `Representer::Create` the way we solved that in the comment API (line 6).

As for the `Representer::Create` code, I put that into `app/concepts/api/v1/thing/representer.rb`.

The representer structure is pretty close to the `Thing::Create` contract we implemented in earlier chapters. However, since we include hypermedia and override some attributes, I prefer having a separate, non-inferred representer⁶².

```

1 module API::V1
2   module Thing
3     module Representer
4       class Create < Roar::Decorator
5         feature Roar::JSON::HAL
6
7         property :name
8         collection :users, embedded: true,
9                     as: :authors,
10                    populator: Reform::Form::Populator::External.new do
11                      property :email
12
13                      link(:self) { api_v1_user_path(represented.id) }
14                    end
15
16        link(:self) { api_v1_thing_path(represented) }
17      end

```

This is really just another representer in the `API::V1` namespace. After marking the representer as a HAL document, I define the `name` property and the `users` collection (line 5-8). The `users` property should be handled as a embedded element (line 8). `Representable` allows to rename properties, and

⁶²Trailblazer allows to infer representers from contracts, and then refine them afterwards. It comes with a nice API to override specific properties, or to inherit and extend. This is helpful if you have huge contracts and representers where redundancy can become a problem. However, in my example, the inheritance would create more code than rewriting the structure. Make sure to [check out the docs](#) if you're planning to used inferred, but refined, representers.

by using `as: :authors`, the users will be keyed as `authors` in the document and parsed from that name (line 9).

By setting the `External` populator, I make sure that when parsing, the form's populator for the `users` fragments is called so I don't have to copy the code to the representer (line 10).

The rest of this representer is old hat and needs no further explanation.

When running the `Create` test, the representer we just implemented will transform the HAL document into the normal nested structure that the form is expecting. It will also take care of renaming `authors` to `users` as the form has no idea about `authors`.

The inherited code from the non-API `Thing::Create` implements the processing and persisting. No work has to be done here.

After the operation's been run, the responder will call `to_json` on it to render the response document. Again, the operation knows that because of the mixed in `Representer` module, and uses representer and model to compile that.

Polymorphic Update

I promised you to speak about polymorphism in APIs. You will remember the `Thing::Update` operation as being polymorph: only signed-in users and admins can update things. Where admins can edit any model, signed-in users can only update their own things.

We implemented this using two subclass `Thing::Update::SignedIn` and `Thing::Update::Admin` which were dispatched using a builder. The good news is: plugging together the already implemented polymorph operations and the API-specific representer is all we have to do right now.

When testing the polymorphic aspects of the update operation, please allow me to discuss the admin case, only. The tests for anonymous users and signed-in users can be found in the repository and are almost identical.

Testing the admin's update goes to `test/api/v1/things_test.rb`.

```
1 describe "PATCH /api/v1/things/1" do
2   let (:thing) do
3     Thing::Create.(
4       {thing: {name: "Lotus", users: [{"email"=> "jacob@trb.org"}]}}
5     ).model
6   end
7
8   it "allows update for admin" do
9     id      = thing.id
10    author_id = thing.users.first.id
11
```

```

12     json = {
13         name:      "Roda",
14         _embedded: { authors: [{ id: "#${author_id}", remove: "1" }] }
15     }.to_json

```

To actually have an asset to update, I use the `Thing::Create` operation as factory where I add a user, too (line 2-6). In the test case, I assign IDs to variables to simplify tests later (line 9-10). I then prepare the JSON document to be sent (line 12-15).

Note that there is a `remove: "1"` property in the document exactly where it would be in the form submission. My goal is to update the existing thing's name to "Roda", and at the same time delete its only user.

While hypermedia formats like JSON API require two requests for that, one to update the model's attributes, one to remove the association member, in HAL there is no restrictions. It is totally up to my server implementation whether or not to respect this as a delete semantic. If you feel like your media format should not support higher-level application semantics, you could simply ignore the `:remove` fragment in the representer⁶³.

```

1 it "allows update for admin" do
2   # ...
3   Session::SignUp::Admin.(user: {email: "admin@trb.org", password: "123456"})
4   authorize("admin@trb.org", "123456")
5
6   patch "/api/v1/things/#{id}/", json

```

Basic Auth

In the next section of the test, I create an admin user using the `Session::SignUp::Admin` operation (line 3).

Using the `authorize` method from Rack-test, the brand-new admin user gets "logged in" (line 4). Don't confuse that with a session login. This will simply set user and password name for the next test request and mark them to be sent in a Basic Auth header, as if you'd filled out the modal dialog box that usually pops open when accessing a Basic Auth-protected website.

And, finally, after all this setup, I issue a PATCH request to the thing resource and pass the JSON document as request body.

Updating an existing resource in a HTTP context usually works by sending a PUT or PATCH request. While a PUT request is meant to replace the existing document with the submitted document, PATCH

⁶³Personally, I am very interested in a richer media format with application semantics that go beyond primitive CRUD operations on URLs, so I decided to leave this as an inspirational feature in the book.

supposedly is a *partial update*. Since I only want to change the name (and remove the author), a PATCH is the way to go.

In Rails, every update is a PATCH, and this ground-breaking convention resulted from a 2-years debate about application semantics, “REST”, and `update_attributes` behavior - a debate that I have gladly ignored.

```

1 it "allows update for admin" do
2   # ...
3   get "/api/v1/things/#{id}"
4
5   last_response.body.must_equal({
6     name:      "Roda",
7     _embedded: { "comments": [] },
8     "_links": { "self": { "href": "/api/v1/things/#{id}" } }
9   }.to_json)
10 end

```

After the PATCH request, I fetch the updated resource using GET (line 3). In the returned document, I make sure the name has changed and there's no users, or authors, referenced anymore (line 5-9).

As for the implementation, I would love to start with a simple route, but we already defined `resources :things` earlier.

After the route comes the endpoint, or, controller, as we call it in Rails. Here's the `update` action in `app/controllers/api/v1/things_controller.rb`.

```

1 module API::V1
2   class ThingsController < ApplicationController
3     def update
4       if request.authorization
5         email, password =
6           ActionController::HttpAuthentication::Basic.
7           user_name_and_password(request)

```

In the `update` action that processes the PATCH, I decided to do authentication manually, for educational purposes. I check if a Basic Auth header was sent in the request using `request.authorization` (line 4).

I then use a Rails function, a very clumsy constant and method, that needs to be spread over three lines to make it format neatly, to retrieve the submitted username and password (line 5-7). Since we authenticate with the user's email, I name the variable accordingly.

```
1 if request.authorization
2   # email, password = ...
3   Session::SignIn.run(session: { email: email, password: password }) do |op|
4     params[:current_user] = op.model
5   end
6 end
7
8 respond Thing::Update, namespace: [:api, :v1]
```

To verify the credentials, I reuse the `Session::SignIn` operation we derived from Tyrant. Please note that I use `run` here to invoke the operation (line 3). With this invocation style, I can make use of the block, which is only executed when the operation was valid. That means the `:current_user` value is only set when the login credentials were valid, because that's what `Session::SignIn` validates (line 4).

An interesting fact here is that I do not set any global variables via Tyrant or Warden. All I do is add the `:current_user` option to the operations params. As you do remember, the operation's builder and policy will look out for this parameter and base further steps on its existence.

Setting global values is always wrong. In the *Authentication* chapter I did this to satisfy Rails' addiction to global variables. Here, we don't use any more Rails code so we don't have to set it. Instead, all that happens after the authentication is the call to `respond`, where the params will be passed to the operation (line 8). So, instead of global state, we explicitly let `respond` pass the params along with the user object to the operation.

Please note that the authentication code could and should totally be abstracted into a separate class, or we could even use Tyrant's HTTP authentication feature⁶⁴. Nevertheless, I find it a lot better to understand going the manual way.

Update Operation

Implementation of the update function you find in `app/concepts/api/v1/thing/operation.rb`.

⁶⁴By the time of writing this chapter, Tyrant did not have the aforementioned feature, yet. As soon as this is shipped, I will reference the documentation here.

```

1 module API::V1
2   module Thing
3     class Update < ::Thing::Update
4       self.builder_class = ::Thing::Create.builder_class
5
6     class Admin < ::Thing::Update::Admin
7       include Trailblazer::Operation::Representer
8       representer Create.representer
9     end
10    end

```

Very little code for a lot of behavior, isn't it? And this is where Trailblazer really starts to take off and shows how inheritance and composition is an incredibly helpful tool when used the right way.

The API::V1::Thing::Update operation inherits contract and process behavior from the original Thing::Update class (line 3). In order to make this operation instantiate the correct sub operation, I copy the builder from Thing::Create into this class (line 4).

Whenever we call API::V1::Thing::Update now, which we do in the update action we just introduced, it will instantiate the SignedIn subclass if a current user's passed in the params, and the Admin subclass if that very user is an admin. If none of that is passed, there will be an authorization exception. This has all been implemented before, and we can simply reference that behavior without having to worry about the internals, again.

What we need to do, though, is to implement the operations that handle the different user types. I will only discuss Admin here as you can find the SignedIn in the repository with very similar code.

All the API::V1::Thing::Update::Admin class needs to do is derive itself from the original one Thing::Update::Admin, include the Representer module and then reference the same representer we used for Create (line 6-9).

That is really everything we need to do. I'll walk you through the entire request stack now.

1. In the test, we instruct the request to carry Basic Auth credentials using `authorize`.
2. We then PATCH a JSON document against `/api/v1/things/:id`, which is meant to change the title of a formerly created Thing instance, and remove its author. The structure of this document is the same as we used for the update form, but embedded in HAL syntax.
3. The update controller action that is being hit by the request uses Rails internals to figure out Basic Auth credentials from the request, authenticates the user and references them as `:current_user` in the params hash. This follows the same API that we used in the *Authentication* chapter.
4. After the manual authentication, the action delegates to the API::V1::Thing::Update operation.
5. This will invoke the operation's builder which will, in our test case, instantiate API::V1::Thing::Update::Admin and pass in the params.

6. Since we reference the representer from the API Create operation, this representer will be used to deserialize the incoming document to the contract. Don't forget, it's a HAL representer, so it understands formating like `_embedded` and will properly write all input to the contract.
7. The contract now represents the changed state from the request. It validates itself, runs populators and eventually will also delete the author from the association.
8. The inherited process method will write the updated Thing and the association changes to the database by saving the model.
9. For the operation, the job is almost done. Now the responder from Rails kicks in and calls `to_json` on the operation, which was about to go to bed, but now uses the representer, again, to serialize the updated model into JSON.
10. The JSON is sent back in the response and we verify it does not contain a user anymore and the title has changed to the name of a very promising Ruby framework.

You might be skeptical as to what would happen if the behavior of the API operation and the normal form-processing operation should be different. The answer is simple: You'd use object-orientation to override the steps you want to change.

This might involve overriding parts of the process method. Or, say you wanted to suppress the deleting of authors, you could override parts of the contract definition and remove the `is_author` field. Even though it looks almost "too easy", the whole conception of operations, builders, representers and contracts is designed to do just that.

Discussion: Polymorphic Operations and APIs

By taking advantage of OOP features, composition and inheritance along with overriding particular hooks, you can reduce code to a minimum where you really want to change specific behavior without running into the trouble of breaking existing code. Every environment, whether that's a form submission for an anonymous author, a signed in user, or an admin user in a JSON environment, is encapsulated in a separate class.

When presenting the OOP design of Trailblazer, many developers criticized this and complained about that "*this will involve too many classes to maintain*". We just learned the opposite: the classes need to exist, that's true, but their code often is only a handful of lines to plug together the required behavior - something that could be done in the background, automatically and convention-driven, if desired.

If suddenly a particular environment has to change, say the signed in user can no longer delete other authors, this is a matter of extending the particular classes that maps to this environment, only. You would change operation and contract code, maybe callbacks, too, in that specific class without having to worry about breaking other code. This is applied OOP the way it was intended to be used, and I do appreciate that.

Perspectives

When presenting Trailblazer at conferences or user groups, I often make the same observation. Either, people are super excited about it. After a talk, someone from the audience asked “*So Trailblazer basically makes everything completely different to Rails? That’s awesome!*”. Those people have felt the pain of Rails’ lack of a high-level architecture.

A funny side-note here is that I’ve met many “juniors” who instantly could see how to place Trailblazer into their existing Rails apps. An eye for architecture sometimes doesn’t require decades of engineering practice - often, your gut feeling is just right.

Then, there’s people who’re absolutely not interested, who’re glad the talk is over.

I am guessing these people have built their own higher abstractions and are happy with it. I have zero interest in attracting happy developers to Trailblazer. If someone’s confident about their architecture, there is no use in convincing them to use your gems just for the sake of convincing them. I had a number of very interesting discussions about how Trailblazer overlaps with other people’s solutions. However, I’ve never seen the integration of service object, form, and presenter, as nicely as in my own framework, but, so be it.

And then, there will be many people criticizing Trailblazer. They will complain “*It has too many objects!*”, or they fearing “*The garbage collection is gonna collapse!*” and the like, or that “*We now have a god class called operation!*” and they will ask for answers to their clever questions.

Usually, we both agree that more abstraction layers are needed. MVC is just not enough.

However, many people don’t understand that objects in Ruby are super cheap. When you parse a JSON document into hashes, there’s thousands and ten-thousands of objects being created, hashes, strings, and so on. Ruby doesn’t care if an object is a cell, an operation, a hash or a string.

The garbage collection doesn’t even know it’s cleaning up a Trailblazer application because the few more objects don’t really make a difference. The opposite is the case, objects can be disposed of quicker because, unlike Rails controllers or models, they are not needed over the entire request.

When it comes to the *God Object* discussion, and I’ve mentioned that earlier in the book, I usually have to explain that orchestration and implementation is not be confused. Just because an operation keeps references to all stakeholders like callbacks, presenter, model, contract, and so on, doesn’t mean it implements these responsibilities - which contradicts the definition of this anti-pattern.

Trailblazer is definitely not meant to be a solution for every programmer on this planet. But what I found out is: many developers, once they reach a certain mindset, are incredibly anxious of accepting other ideas. This includes myself. It is very hard and challenging to give up your own approach and allow other’s thoughts to help solving your problem.

A good way is to talk to each other, identify what issues need be get solved and then discuss what patterns might be helpful. There's many stories I could tell now, for example, why my code has changed radically from a very fuzzy object to functional, strict APIs, where objects often just have one public method. But this is content for another book.

The cool thing about Trailblazer is: you don't have to sell it at once. It can be step-wise introduced, and developers who were initially protesting against it might get convinced by seeing how and where it actually helps.

This Book

Writing this book was an incredible experience. What might sound to you like a classic author's platitude I mean for real.

The traction that Trailblazer got during the writing of this book was beyond any expectations of mine. I did absolutely *not* believe in the idea that entire companies will base their softwares on the Trailblazer framework at that early stage. I learned that teams used Trailblazer with Webmachine, with Roda, or to refactor classic Rails apps.

It was a very humbling feeling, but at the same time, I got stressed. Trailblazer, in my eyes, still had a long way to go, I wanted to finalize the embraced gems, refactor problematic code I've hated for years, and so on. And yet, entire products were using those gems.

Most teams are still using Trailblazer at the present day. I've heard of a few who've given up due to the - understandably - bumpy ride, but those who stick do not seem to regret it. They now enjoy the maturity and stability of an evolving framework.

Most users report that Trailblazer has dramatically improved their code structure. By introducing *Operation* and making every public function of the application an enclosed object, teams are forced to communicate in APIs - a fantastic restriction in software development.

Also, the operation container is easy to test, and no longer do you have to think about what to test where: Logic is tested via an operation test, wiring and view in integration tests, and you're done.

The integration of contract in the operation, along with processing logic made developers think in layers. Suddenly, the melting pot of models became different steps: deserialize to a contract, validate the contract, run your business code based on this contract twin, and then let the twin persist the data to your ORM models.

Why Not a New Framework?

Often, I get asked why I don't write a new framework for Ruby instead of "*having to integrate Trailblazer*" with all the existing ones.

My question is always: "*Why would I?*" There's a bunch of very promising frameworks out there, many of them battle-tested, fast, cleanly implemented without making monstrous assumptions

about your objects. The core teams of those projects have focused years on providing what I call an *infrastructure framework*: they have spent many thoughts on library file organization, routing, dispatching, automatic tasks, asset pipelining, migrations and bindings to various ORMs, test integration and many things more.

It would be both time-consuming and frustrating to try to compete with existing solutions. Look at it from another perspective: Trailblazer is good at the business, at implementing the high-level architecture of web applications. This is where most frameworks - no offence - have rather half-baked solutions for deserialization, validation or post-processing.

Trailblazer is definitely the strongest framework for those higher level tasks.

This is, because we focused on those architectural layers exclusively, while neglecting the infrastructure. Of course, we knew that there's many other projects to pick from that provide us with a dispatching logic we could never have rewritten in such maturity.

In other words: I prefer to focus on making one thing good, instead of trying to deliver a all-in-one solution. Another nice side-effect is: the more you decouple your work from actual frameworks, the better your APIs will get.

And now, developers have the choice of what to pick and where to use their choice with Trailblazer!

Rails

Rails has brought together many many people, companies and projects that rely on this framework. The eco-system that has evolved around it is incredible, and it feels great to be a part of it.

In the last couple of years, however, Rails has lost leadership. Personally, I can't identify who's pushing for change, innovation, or critical thinking anymore. There's no big picture. Little features like ActiveJob or ActionCable are added here and there, none of them with structural impact to make us write better software.

Internally, Rails is standing still: it's impossible to refactor parts without having to change concepts or API. This desperately needed *change* seems to be a no-go for the ruling party - a stubborn attitude that will navigate Rails into a not meaningless, but no more meaningful existence in the next year or two.

Personally, I am not too interested in using or improving Rails anymore. I find everything about the Rails Way is wrong, which I hope to have pointed out in this primer. In hindsight, I've learned many many lessons through Rails, and I do appreciate that. The pain and trouble I've had fighting Rails at basically every point of writing a non-Basecamp application is definitely outweighing the benefits I got from a handful of features like migrations or test integration.

I wish I had better news regarding Rails, that it is on the right path, but the amount of smaller, more innovative companies that are switching to newer frameworks is saying the opposite.

Webmachine

Webmachine's philosophy is to not provide a DSL, but let users inherit classes and then overwrite behavioral methods - a concept I do appreciate a lot. For a long time, Webmachine was the only framework that implemented the HTTP specification correctly, and it might still be able to claim that.

I've never used it myself except for a brief session side by side with the author Sean Cribbs, where we connected Webmachine with Roar. I know of several companies, though, that use Webmachine and implement every Resource with Trailblazer operations, exactly how we did it in the *Hypermedia APIs* chapters but without the Rails noise.

Grape

Grape is a framework exclusively to build hypermedia APIs. It comes with a simple, pluggable routing architecture where endpoints implement the representation of the resources. Grape provides its own implementation of rendering, deserialization and validation.

Nevertheless, Grape also allows replacing huge parts with Trailblazer. I am currently working on an API project where Grape provides the routing, only. The entire high-level architecture is implemented in Trailblazer operations. This is one of the most beautiful projects I've ever worked on.

ROM

The ROM project is not really a web framework, even though it is extremely helpful in this environment. It helps modelling persistent objects to arbitrary data structures. ROM's very modular architecture then allows to hook processing and validation logic into that mapping task, making it a very functional-oriented tool that minimizes state.

It works perfectly in Trailblazer's operations and can replace the persistence layer or even higher processing steps. While Trailblazer provides the higher application structure, deserialization and validation, ROM can act as the ideal counterpart for everything that comes after that. I've seen ROM and Trailblazer play together in a few friends' projects and can't wait to start playing with it myself.

Roda

Main focus of Roda is a routing API that allows defining and plugging in your endpoints. Roda, just like Grape and Lotus, doesn't do nasty Ruby work-around but is implemented very cleanly and fast. It also comes with a huge selection of plugins, from serializers to partials and template support.

So far, I have only heard from other developer teams about how nicely Roda and Trailblazer interact, since the operation can be hooked into the routing tree to perform the business logic. Apparently, Cells also plays nicely with this stack and allows a rich UI layer on top of an operation-backed routing tree. It is needless to say that I am intrigued by this partnership and need to try it out in the near future.

Lotus

Lotus is a relatively new framework that claims to be “*a complete web framework*”. It is a bit too early to comment on its high-level architecture, which I am missing in Lotus at the moment. However, exactly that makes it a perfect partner for Trailblazer.

Lotus’ implementation is incredibly clean, free of monkey-patching and the team focuses on integrating more low level tasks like migrations, asset pipeline or tasks. Its endpoint `Lotus::Action` is where you can hook in Trailblazer - where Lotus handles the transportation, routing and infrastructure, Trailblazer can do what it’s best at: the business.

During the writing of this book I’ve learned from others that this works pretty neatly. Since Trailblazer doesn’t have any Rails bindings, engineers started using Lotus and operations together and seem to be very happy. Another framework on my list to try and get excited!

Trailblazer

A few users have noticed when they started playing with Trailblazer that it felt like they were “only moving code around”. But even that felt better, since the code moved from messy god-classes into small, maintainable containers. Of course, Trailblazer still requires you to program things, it can’t magically write your business logic for you.

However, the way it’s architected is designed to streamline common web application development, at a level much higher than Rails or other existing infrastructure frameworks. And once those moaning users learned how to take advantage of compositions, the Reform contract, twins and callbacks, they all reported that their code base is shrinking and the tests grow - a good sign, I’d say!

My advice is: don’t expect to learn everything about Trailblazer in a few days. It took almost 10 years to invent, write and evolve all this technology, so relax and let it sink in.

I’m absolutely confident that Trailblazer will help many companies, teams and freelancers to build better, faster, solid and sustainable software with Ruby, and I’m excited about the future and what stories you’re going to tell me.

Thanks for blazing trails with me - and *See You On The Trail!*

Nick