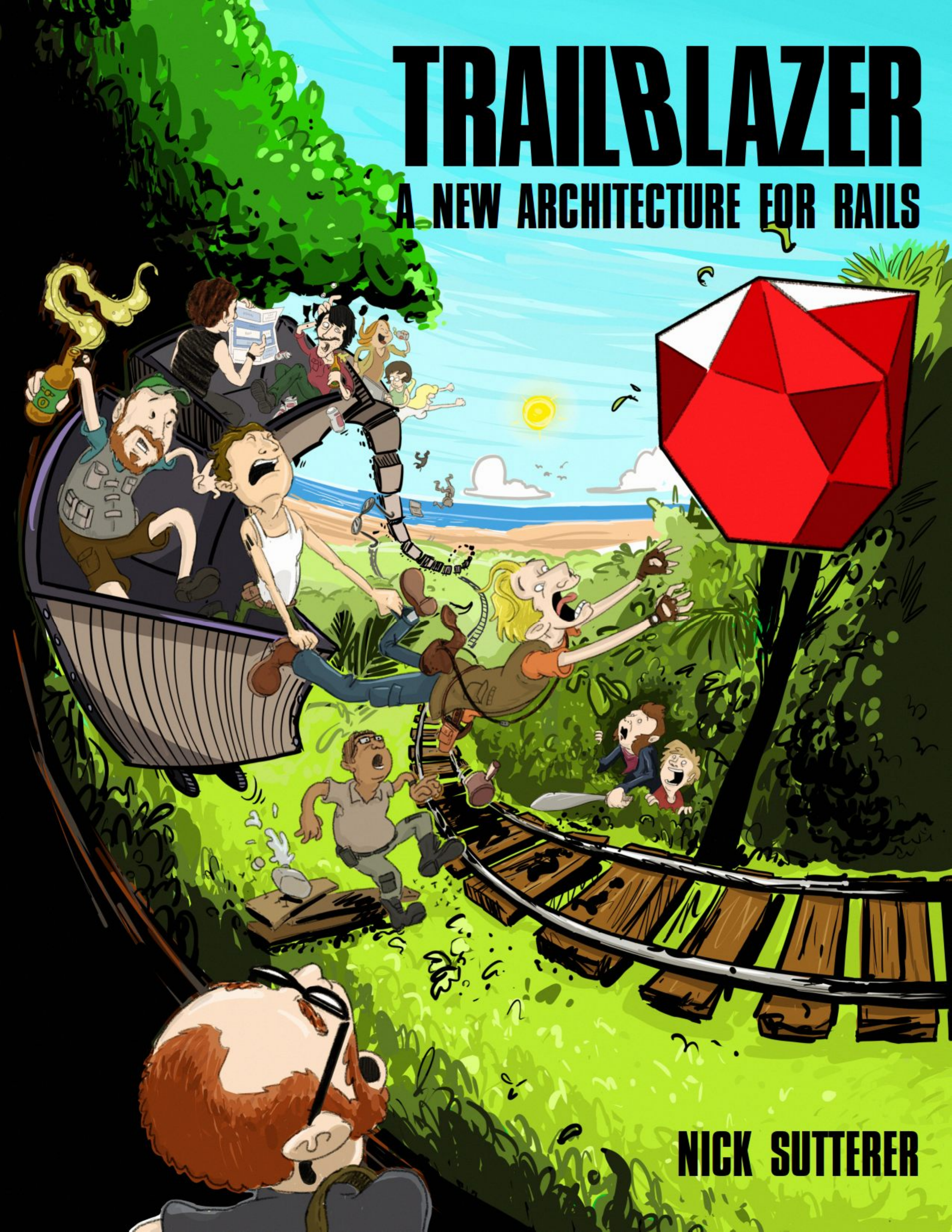
TRAILBLAZER

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 2016-06-22

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

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 ingenius 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 is used in other frameworks like Roda, Sinatra or Grape and make more people happy. Oh, excuse me, "Sinatra is not a framework"[1].

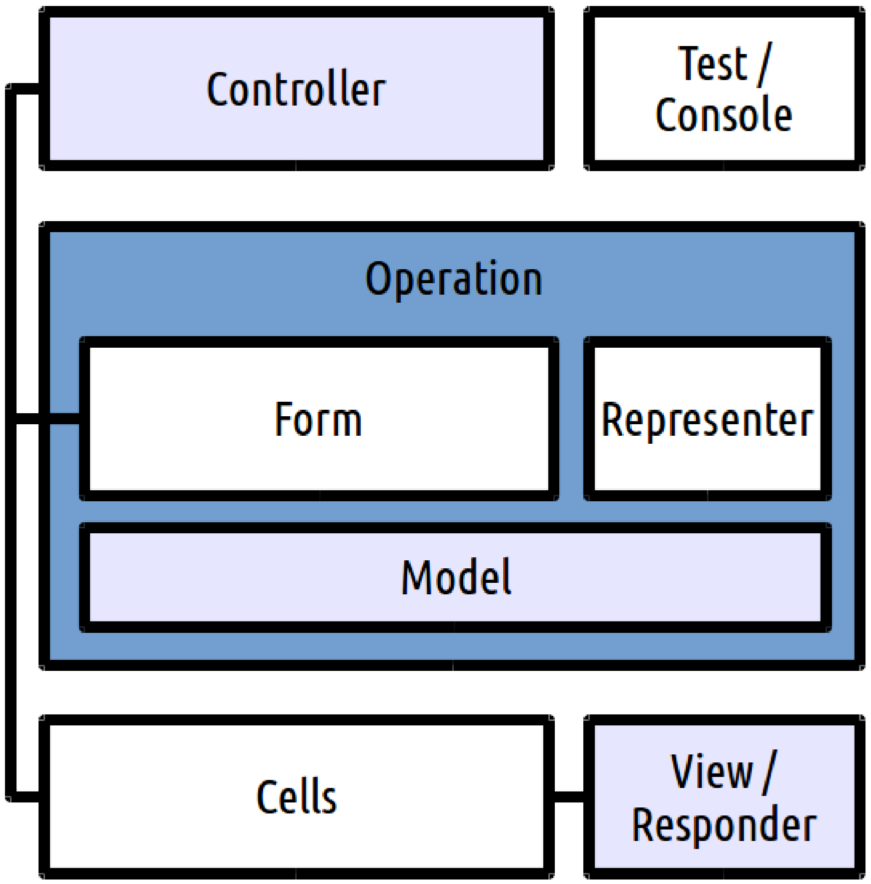
This loose cohesion makes Trailblazer a, what I call, "non intrusive framework". As stated in the README it does not missionize you. You decide where and how much Trailblazer you want and where the Rails Way is sufficient. They don't interfere with each other.

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.

[1]Konstantin 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.

```
class Rating < ActiveRecord::Base
  belongs_to :thing
  belongs_to :author, through: :user
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[2] originally does include "domain logic".

[2]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.

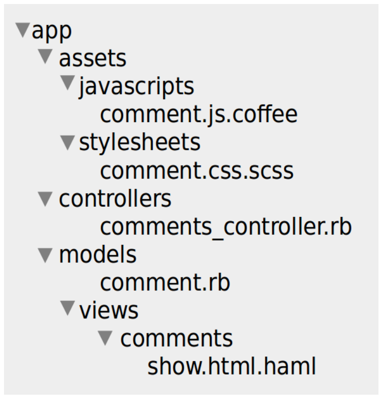
The business logic for both reading and writing along with validations is abstracted into new layers: Operation, Form and Twin are patterns that Trailblazer brings to your architecture.

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.



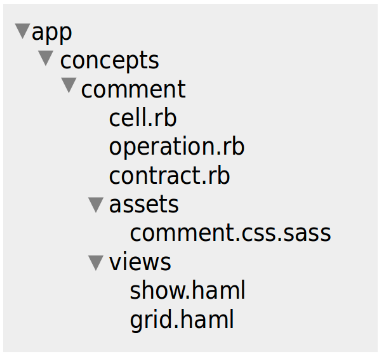
```
▼app
  ▼assets
    ▼javascripts
       comment.js.coffee
    ▼stylesheets
       comment.css.scss
  ▼controllers
       comments_controller.rb
  ▼models
       comment.rb
    ▼views
      ▼comments
          show.html.haml
```

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.



```
▼app
  ▼concepts
    ▼comment
        cell.rb
        operation.rb
        contract.rb
      ▼assets
          comment.css.sass
      ▼views
          show.haml
          grid.haml
```

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.

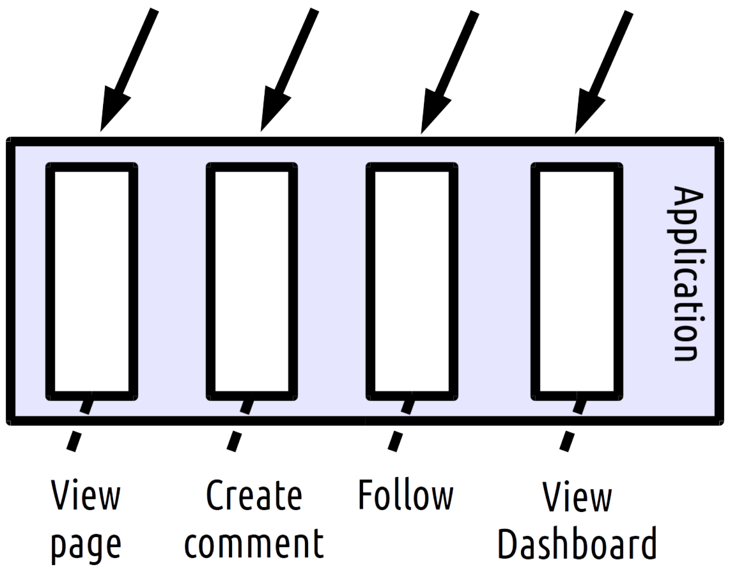
Using a web application means you send requests, that queries or changes application state, then you get a response. The browser hides that from you, but behind the scenes this is exactly what happens. I want you to think of every *function* of a web application as a request-response cycle. It's not 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.

```ruby
def create
  @comment = Comment.new { |c| c.author = current_profile }
  # .. more stuff
end
```

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

```ruby
def create
  @project = Project.find(params[:project_id])
  @task = @project.tasks.build(params[:task])
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.

```ruby
def create
  # ..
  if @project.valid? and @task.valid?
    redirect_to projects_path
  end
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.

```ruby
class CommentsController < ApplicationController
  def new
    present Comment::Create
  end

  def create
    run Comment::Create
  end
  # ..
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 going to 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.

```ruby
class Comment < ActiveRecord::Base
  class Create < Trailblazer::Operation
    def process(params)
      # ..
    end
  end
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*.

```ruby
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.

```ruby
def create
  Comment::Create.(params)
end
```

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

```ruby
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.

```ruby
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 Traiblazer, 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 going to 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 is 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 explicitely 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.

```ruby
class Comment < ActiveRecord::Base
  class Create < Trailblazer::Operation
    def process(params)
      Comment.create(params)
    end
  end
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.

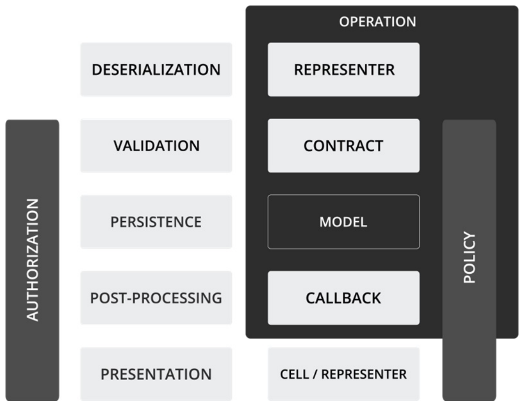
The operation here acts as an *orchestrator* dispatching between the different layers of Trailblazer. The layers are all separate objects that have no idea about the context they're being used in, or what an operation is.

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 Reform[3] form class. If you've used this gem before, you will recognize the API.

```ruby
class Create < Trailblazer::Operation
  contract do
    property :body,   validates: {presence: true}
    property :author, validates: {association: true}
  end

  def process(params)
  # ..
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 `ActiveModel::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.

```ruby
class Create < Trailblazer::Operation
  contract do
    # ..
  end

  def process(params)
    @model = Comment.new

    validate(params[:comment]) do |f|
      # process the data here
    end
  end
end
```

[3] 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 going to 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.

```ruby
def process(params)
  @model = Comment.new

  validate(params[:comment]) do |f|
    f.save

    notify!
    log!
  end
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 going to 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 going to 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 going to discover in chapter 8.

Operations provide a clean way to group callbacks. Instead of adding ActiveRecord callbacks with conditionals you explicitely 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 going to 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.

```ruby
class Update < Create
  action :update
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.

```ruby
class Update < Create
  action :update

  contract do
    property :author, readonly: true # make author read-only for update.
  end
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 going to love.

```ruby
class Create
  class JSON < self
    include Representer

    representer do
      property :author do
        property :name, as: :fullName
      end
    end
  end
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.

```ruby
def create
  Comment::Create.(params)
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.

```ruby
class Create < Trailblazer::Operation
  builds ->(params) do
    Moderated if params[:current_user].needs_moderation?
  end

  class Moderated < self
    # ..
  end
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.

```ruby
Comment::Create.(current_user: admin) #=> Comment::Create
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.

```
class Create < Trailblazer::Operation
  builds ->(params) do
    user = params[:current_user]

    return Create    if user.admin?
    return Moderated if user.can_post?
    deny!
  end

  # ..
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 cancancan, 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.

```
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.

```
class Create < Trailblazer::Operation
  policy do |params|
    Permit.can?(:update, model)
  end

  # ..
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[4], 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 the 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[5] in this book to authenticate users and cookie managment.

```
class Session::SignIn < Trailblazer::Operation
  def process(params)
    Tyrant::SignIn.(params)

    mark_online_activity!
  end
```

[4]https://github.com/plataformatec/devise
[5]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[6], rabl[7] or ActiveModel::Serializer[8].

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 going to use a lot in this book.

Representers in Trailblazer come from the Representable[9] 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.

[6]https://github.com/rails/jbuilder
[7]https://github.com/nesquena/rabl
[8]https://github.com/rails-api/active_model_serializers
[9]https://github.com/apotonick/representable

```ruby
class CommentRepresenter < Representable::Decorator
  include JSON

  property :body
  property :author do
    property :email
  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.

```ruby
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.

```ruby
CommentRepresenter.new(comment).from_json("{\"body\": \"Really?\",..}")
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.

```ruby
class Comment < ActiveRecord::Base
  class Create < Trailblazer::Operation
    # .. all the code from earlier

    class JSON < self
      include Representer
    end
  end
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.

```ruby
def create
  respond Comment::Create
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.

```ruby
class Create < Trailblazer::Operation
  # ..

  class JSON < self
    representer do
      property :related_comments do
        property :id
      end
    end
  end
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 Roar[10] 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.

```
class JSON < self
  representer do
    include Roar::JSON::HAL

    link :self do
      comment_path(represented.id)
    end
  end
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.

```
representer do
  include Roar::JSON::JSONAPI

  # ..
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

[10] 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 representer 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.

```ruby
def create
  run Comment::Create do |op|
    # valid operation
    return redirect_to comment_path(@model)
  end

  render action: :new # has access to @model
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 Cells[11] 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 going to use them in views and controllers to replace a helper/partial-mess with a decent view component.

```
%h1 Latest Comment

= 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 going to clean up that mess?

[11] https://github.com/apotonick/cells

```ruby
class Comment::Cell < Cell::ViewModel
  def show
    render
  end
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.

```haml
#recent
  = body
  .author
    = 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.

```ruby
class Comment::Cell < Cell::ViewModel
  def show
    render
  end

private
  def body
    model.body
  end

  def author_link
    link_to model.author.name, author_path(model.author)
  end
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.

```
class Comment::Cell < Cell::ViewModel
  def show
    render
  end

private
  properties :body, :author

  def author_link
    link_to author.name, author_path(author)
  end
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 going to 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 critizise 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 going to learn everything about helpers, object-oriented partials, reusable templates and polymorphic views in this book.

Twin

While most of the logic is going to 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.

```ruby
class Comment::Presentation < Disposable::Twin
  property :body
  property :thing

  def public?
    thing.public?
  end

  # ..
end
```

A twin decorator makes logic reusable and we're going to 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.

```ruby
class Comment::Opinion < Disposable::Twin
  include Composition

  property :body,  on: :comment
  property :email, on: :author
end
```

When instantiating the twin needs both objects.

```ruby
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.

```
test "POST /comments" do
  post "/comments", {comment: { body: "Awesome!" }}.to_json
  assert_redirect_to comment_path(operation.model)
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 going to test everything that could go wrong and right.

```
describe "Create" do
  it "is valid" do
    op = Comment::Create.(comment: {body: "Awesome!"})

    op.model.body.must_equal "Awesome!"
    op.model.persisted?.must_equal true
    op.message.must_equal "Comment for Traiblazer was created!"
  end
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.

```
describe "Respond" do
  it "[valid]" do
    comment = Comment::Create.(comment: {body: "Awesome!"}).model

    op = Comment::Respond.(id: comment.id, comment: {..})
  end
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.

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

it do
  cell("comment/row", comment).must_have_css("li.comment")
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 explicitely 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.