DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Getting Started with Engines

In this guide you will learn about engines and how they can be used to provide additional functionality to their host applications through a clean and very easy-to-use interface.

After reading this guide, you will know:

- What makes an engine.
- How to generate an engine.
- How to build features for the engine.
- How to hook the engine into an application.
- How to override engine functionality in the application.
- How to avoid loading Rails frameworks with Load and Configuration Hooks.

What are Engines?

Engines can be considered miniature applications that provide functionality to their host applications. A Rails application is actually just a "supercharged" engine, with the `Rails::Application` class inheriting a lot of its behavior from `Rails::Engine`.

Therefore, engines and applications can be thought of as almost the same thing, just with subtle differences, as you'll see throughout this guide. Engines and applications also share a common structure.

Engines are also closely related to plugins. The two share a common `lib` directory structure, and are both generated using the `rails plugin new` generator. The difference is that an engine is considered a "full plugin" by Rails (as indicated by the `--full` option that's passed to the generator command). We'll actually be using the `--mountable` option here, which includes all the features of `--full`, and then some. This guide will refer to these "full plugins" simply as "engines" throughout. An engine **can** be a plugin, and a plugin **can** be an engine.

The engine that will be created in this guide will be called "blorgh". This engine will provide blogging functionality to its host applications, allowing for new articles and comments to be created. At the beginning of this guide, you will be working solely within the engine itself, but in later sections you'll see how to hook it into an application.

Engines can also be isolated from their host applications. This means that an application is able to have a path provided by a routing helper such as `articles_path` and use an engine that also provides a path also called `articles_path`, and the two would not clash. Along with this, controllers, models and table names are also namespaced. You'll see how to do this later in this guide.

It's important to keep in mind at all times that the application should **always** take precedence over its engines. An application is the object that has final say in what goes on in its environment. The engine should only be enhancing it, rather than changing it drastically.

To see demonstrations of other engines, check out Devise, an engine that provides authentication for its parent applications, or Thredded, an engine that provides forum functionality. There's also Spree which provides an e-commerce platform, and Refinery CMS, a CMS engine.

Finally, engines would not have been possible without the work of James Adam, Piotr Sarnacki, the Rails Core Team, and a number of other people. If you ever meet them, don't forget to say thanks!

Generating an Engine

To generate an engine, you will need to run the plugin generator and pass it options as appropriate to the need. For the "blorgh" example, you will need to create a "mountable" engine, running this command in a terminal:

```
$ rails plugin new blorgh --mountable
```

The full list of options for the plugin generator may be seen by typing:

```
$ rails plugin --help
```

The `--mountable` option tells the generator that you want to create a "mountable" and namespace-isolated engine. This generator will provide the same skeleton structure as would the `--full` option. The `--full` option tells the generator that you want to create an engine, including a skeleton structure that provides the following:

- An `app` directory tree
- A `config/routes.rb` file:

  ```
  Rails.application.routes.draw do
  end
  ```

- A file at `lib/blorgh/engine.rb`, which is identical in function to a standard Rails application's `config/application.rb` file:

  ```
  module Blorgh
    class Engine < ::Rails::Engine
    end
  end
  ```

The `--mountable` option will add to the `--full` option:

- Asset manifest files (`blorgh_manifest.js` and `application.css`)
- A namespaced `ApplicationController` stub
- A namespaced `ApplicationHelper` stub
- A layout view template for the engine
- Namespace isolation to `config/routes.rb` :

  ```
  Blorgh::Engine.routes.draw do
  end
  ```

- Namespace isolation to `lib/blorgh/engine.rb` :

  ```
  module Blorgh
    class Engine < ::Rails::Engine
  ```

```
    isolate_namespace Blorgh
  end
end
```

Additionally, the `--mountable` option tells the generator to mount the engine inside the dummy testing application located at `test/dummy` by adding the following to the dummy application's routes file at `test/dummy/config/routes.rb`:

```
mount Blorgh::Engine => "/blorgh"
```

Inside an Engine

Critical Files

At the root of this brand new engine's directory lives a `blorgh.gemspec` file. When you include the engine into an application later on, you will do so with this line in the Rails application's `Gemfile`:

```
gem 'blorgh', path: 'engines/blorgh'
```

Don't forget to run `bundle install` as usual. By specifying it as a gem within the `Gemfile`, Bundler will load it as such, parsing this `blorgh.gemspec` file and requiring a file within the `lib` directory called `lib/blorgh.rb`. This file requires the `blorgh/engine.rb` file (located at `lib/blorgh/engine.rb`) and defines a base module called `Blorgh`.

```
require "blorgh/engine"

module Blorgh
end
```

TIP: Some engines choose to use this file to put global configuration options for their engine. It's a relatively good idea, so if you want to offer configuration options, the file where your engine's `module` is defined is perfect for that. Place the methods inside the module and you'll be good to go.

Within `lib/blorgh/engine.rb` is the base class for the engine:

```
module Blorgh
  class Engine < ::Rails::Engine
    isolate_namespace Blorgh
  end
end
```

By inheriting from the `Rails::Engine` class, this gem notifies Rails that there's an engine at the specified path, and will correctly mount the engine inside the application, performing tasks such as adding the `app` directory of the engine to the load path for models, mailers, controllers, and views.

The `isolate_namespace` method here deserves special notice. This call is responsible for isolating the controllers, models, routes, and other things into their own namespace, away from similar components inside the application. Without this, there is a possibility that the engine's components could "leak" into the application,

causing unwanted disruption, or that important engine components could be overridden by similarly named things within the application. One of the examples of such conflicts is helpers. Without calling `isolate_namespace`, the engine's helpers would be included in an application's controllers.

NOTE: It is **highly** recommended that the `isolate_namespace` line be left within the `Engine` class definition. Without it, classes generated in an engine **may** conflict with an application.

What this isolation of the namespace means is that a model generated by a call to `bin/rails generate model`, such as `bin/rails generate model article`, won't be called `Article`, but instead be namespaced and called `Blorgh::Article`. In addition, the table for the model is namespaced, becoming `blorgh_articles`, rather than simply `articles`. Similar to the model namespacing, a controller called `ArticlesController` becomes `Blorgh::ArticlesController` and the views for that controller will not be at `app/views/articles`, but `app/views/blorgh/articles` instead. Mailers, jobs and helpers are namespaced as well.

Finally, routes will also be isolated within the engine. This is one of the most important parts about namespacing, and is discussed later in the Routes section of this guide.

`app` Directory

Inside the `app` directory are the standard `assets`, `controllers`, `helpers`, `jobs`, `mailers`, `models`, and `views` directories that you should be familiar with from an application. We'll look more into models in a future section, when we're writing the engine.

Within the `app/assets` directory, there are the `images` and `stylesheets` directories which, again, you should be familiar with due to their similarity to an application. One difference here, however, is that each directory contains a sub-directory with the engine name. Because this engine is going to be namespaced, its assets should be too.

Within the `app/controllers` directory there is a `blorgh` directory that contains a file called `application_controller.rb`. This file will provide any common functionality for the controllers of the engine. The `blorgh` directory is where the other controllers for the engine will go. By placing them within this namespaced directory, you prevent them from possibly clashing with identically-named controllers within other engines or even within the application.

NOTE: The `ApplicationController` class inside an engine is named just like a Rails application in order to make it easier for you to convert your applications into engines.

NOTE: If the parent application runs in `classic` mode, you may run into a situation where your engine controller is inheriting from the main application controller and not your engine's application controller. The best way to prevent this is to switch to `zeitwerk` mode in the parent application. Otherwise, use `require_dependency` to ensure that the engine's application controller is loaded. For example:

```
# ONLY NEEDED IN `classic` MODE.
require_dependency "blorgh/application_controller"

module Blorgh
  class ArticlesController < ApplicationController
    # ...
  end
end
```

WARNING: Don't use `require` because it will break the automatic reloading of classes in the development environment - using `require_dependency` ensures that classes are loaded and unloaded in the correct manner.

Just like for `app/controllers`, you will find a `blorgh` subdirectory under the `app/helpers`, `app/jobs`, `app/mailers` and `app/models` directories containing the associated `application_*.rb` file for gathering common functionalities. By placing your files under this subdirectory and namespacing your objects, you prevent them from possibly clashing with identically-named elements within other engines or even within the application.

Lastly, the `app/views` directory contains a `layouts` folder, which contains a file at `blorgh/application.html.erb`. This file allows you to specify a layout for the engine. If this engine is to be used as a stand-alone engine, then you would add any customization to its layout in this file, rather than the application's `app/views/layouts/application.html.erb` file.

If you don't want to force a layout on to users of the engine, then you can delete this file and reference a different layout in the controllers of your engine.

`bin` Directory

This directory contains one file, `bin/rails`, which enables you to use the `rails` sub-commands and generators just like you would within an application. This means that you will be able to generate new controllers and models for this engine very easily by running commands like this:

```
$ bin/rails generate model
```

Keep in mind, of course, that anything generated with these commands inside of an engine that has `isolate_namespace` in the `Engine` class will be namespaced.

`test` Directory

The `test` directory is where tests for the engine will go. To test the engine, there is a cut-down version of a Rails application embedded within it at `test/dummy`. This application will mount the engine in the `test/dummy/config/routes.rb` file:

```
Rails.application.routes.draw do
  mount Blorgh::Engine => "/blorgh"
end
```

This line mounts the engine at the path `/blorgh`, which will make it accessible through the application only at that path.

Inside the test directory there is the `test/integration` directory, where integration tests for the engine should be placed. Other directories can be created in the `test` directory as well. For example, you may wish to create a `test/models` directory for your model tests.

Providing Engine Functionality

The engine that this guide covers provides submitting articles and commenting functionality and follows a similar thread to the Getting Started Guide, with some new twists.

NOTE: For this section, make sure to run the commands in the root of the `blorgh` engine's directory.

Generating an Article Resource

The first thing to generate for a blog engine is the `Article` model and related controller. To quickly generate this, you can use the Rails scaffold generator.

```
$ bin/rails generate scaffold article title:string text:text
```

This command will output this information:

```
invoke  active_record
create    db/migrate/[timestamp]_create_blorgh_articles.rb
create    app/models/blorgh/article.rb
invoke    test_unit
create      test/models/blorgh/article_test.rb
create      test/fixtures/blorgh/articles.yml
invoke  resource_route
 route    resources :articles
invoke  scaffold_controller
create    app/controllers/blorgh/articles_controller.rb
invoke    erb
create      app/views/blorgh/articles
create      app/views/blorgh/articles/index.html.erb
create      app/views/blorgh/articles/edit.html.erb
create      app/views/blorgh/articles/show.html.erb
create      app/views/blorgh/articles/new.html.erb
create      app/views/blorgh/articles/_form.html.erb
invoke    test_unit
create      test/controllers/blorgh/articles_controller_test.rb
create      test/system/blorgh/articles_test.rb
invoke    helper
create      app/helpers/blorgh/articles_helper.rb
invoke      test_unit
```

The first thing that the scaffold generator does is invoke the `active_record` generator, which generates a migration and a model for the resource. Note here, however, that the migration is called `create_blorgh_articles` rather than the usual `create_articles`. This is due to the `isolate_namespace` method called in the `Blorgh::Engine` class's definition. The model here is also namespaced, being placed at `app/models/blorgh/article.rb` rather than `app/models/article.rb` due to the `isolate_namespace` call within the `Engine` class.

Next, the `test_unit` generator is invoked for this model, generating a model test at `test/models/blorgh/article_test.rb` (rather than `test/models/article_test.rb`) and a fixture at `test/fixtures/blorgh/articles.yml` (rather than `test/fixtures/articles.yml`).

After that, a line for the resource is inserted into the `config/routes.rb` file for the engine. This line is simply `resources :articles`, turning the `config/routes.rb` file for the engine into this:

```
Blorgh::Engine.routes.draw do
  resources :articles
end
```

Note here that the routes are drawn upon the `Blorgh::Engine` object rather than the `YourApp::Application` class. This is so that the engine routes are confined to the engine itself and can be mounted at a specific point as shown in the test directory section. It also causes the engine's routes to be isolated from those routes that are within the application. The Routes section of this guide describes it in detail.

Next, the `scaffold_controller` generator is invoked, generating a controller called `Blorgh::ArticlesController` (at `app/controllers/blorgh/articles_controller.rb`) and its related views at `app/views/blorgh/articles`. This generator also generates tests for the controller (`test/controllers/blorgh/articles_controller_test.rb` and `test/system/blorgh/articles_test.rb`) and a helper (`app/helpers/blorgh/articles_helper.rb`).

Everything this generator has created is neatly namespaced. The controller's class is defined within the `Blorgh` module:

```
module Blorgh
  class ArticlesController < ApplicationController
    # ...
  end
end
```

NOTE: The `ArticlesController` class inherits from `Blorgh::ApplicationController`, not the application's `ApplicationController`.

The helper inside `app/helpers/blorgh/articles_helper.rb` is also namespaced:

```
module Blorgh
  module ArticlesHelper
    # ...
  end
end
```

This helps prevent conflicts with any other engine or application that may have an article resource as well.

You can see what the engine has so far by running `bin/rails db:migrate` at the root of our engine to run the migration generated by the scaffold generator, and then running `bin/rails server` in `test/dummy`. When you open `http://localhost:3000/blorgh/articles` you will see the default scaffold that has been generated. Click around! You've just generated your first engine's first functions.

If you'd rather play around in the console, `bin/rails console` will also work just like a Rails application. Remember: the `Article` model is namespaced, so to reference it you must call it as `Blorgh::Article`.

```
irb> Blorgh::Article.find(1)
=> #<Blorgh::Article id: 1 ...>
```

One final thing is that the `articles` resource for this engine should be the root of the engine. Whenever someone goes to the root path where the engine is mounted, they should be shown a list of articles. This can be made to happen if this line is inserted into the `config/routes.rb` file inside the engine:

```
root to: "articles#index"
```

Now people will only need to go to the root of the engine to see all the articles, rather than visiting `/articles`. This means that instead of `http://localhost:3000/blorgh/articles`, you only need to go to `http://localhost:3000/blorgh` now.

Generating a Comments Resource

Now that the engine can create new articles, it only makes sense to add commenting functionality as well. To do this, you'll need to generate a comment model, a comment controller, and then modify the articles scaffold to display comments and allow people to create new ones.

From the engine root, run the model generator. Tell it to generate a `Comment` model, with the related table having two columns: an `article_id` integer and `text` text column.

```
$ bin/rails generate model Comment article_id:integer text:text
```

This will output the following:

```
invoke  active_record
create    db/migrate/[timestamp]_create_blorgh_comments.rb
create    app/models/blorgh/comment.rb
invoke    test_unit
create      test/models/blorgh/comment_test.rb
create      test/fixtures/blorgh/comments.yml
```

This generator call will generate just the necessary model files it needs, namespacing the files under a `blorgh` directory and creating a model class called `Blorgh::Comment`. Now run the migration to create our blorgh_comments table:

```
$ bin/rails db:migrate
```

To show the comments on an article, edit `app/views/blorgh/articles/show.html.erb` and add this line before the "Edit" link:

```
<h3>Comments</h3>
<%= render @article.comments %>
```

This line will require there to be a `has_many` association for comments defined on the `Blorgh::Article` model, which there isn't right now. To define one, open `app/models/blorgh/article.rb` and add this line into the model:

```
has_many :comments
```

Turning the model into this:

```
module Blorgh
  class Article < ApplicationRecord
    has_many :comments
  end
end
```

NOTE: Because the `has_many` is defined inside a class that is inside the `Blorgh` module, Rails will know that you want to use the `Blorgh::Comment` model for these objects, so there's no need to specify that using the `:class_name` option here.

Next, there needs to be a form so that comments can be created on an article. To add this, put this line underneath the call to `render @article.comments` in `app/views/blorgh/articles/show.html.erb`:

```
<%= render "blorgh/comments/form" %>
```

Next, the partial that this line will render needs to exist. Create a new directory at `app/views/blorgh/comments` and in it a new file called `_form.html.erb` which has this content to create the required partial:

```
<h3>New comment</h3>
<%= form_with model: [@article, @article.comments.build] do |form| %>
  <p>
    <%= form.label :text %><br>
    <%= form.text_area :text %>
  </p>
  <%= form.submit %>
<% end %>
```

When this form is submitted, it is going to attempt to perform a `POST` request to a route of `/articles/:article_id/comments` within the engine. This route doesn't exist at the moment, but can be created by changing the `resources :articles` line inside `config/routes.rb` into these lines:

```
resources :articles do
  resources :comments
end
```

This creates a nested route for the comments, which is what the form requires.

The route now exists, but the controller that this route goes to does not. To create it, run this command from the engine root:

```
$ bin/rails generate controller comments
```

This will generate the following things:

```
create  app/controllers/blorgh/comments_controller.rb
invoke  erb
 exist    app/views/blorgh/comments
invoke  test_unit
create    test/controllers/blorgh/comments_controller_test.rb
invoke  helper
create    app/helpers/blorgh/comments_helper.rb
invoke    test_unit
```

The form will be making a `POST` request to `/articles/:article_id/comments`, which will correspond with the `create` action in `Blorgh::CommentsController`. This action needs to be created, which can be done by putting the following lines inside the class definition in `app/controllers/blorgh/comments_controller.rb`:

```
def create
  @article = Article.find(params[:article_id])
  @comment = @article.comments.create(comment_params)
  flash[:notice] = "Comment has been created!"
  redirect_to articles_path
end

private
  def comment_params
    params.require(:comment).permit(:text)
  end
```

This is the final step required to get the new comment form working. Displaying the comments, however, is not quite right yet. If you were to create a comment right now, you would see this error:

```
Missing partial blorgh/comments/_comment with {:handlers=>[:erb, :builder],
:formats=>[:html], :locale=>[:en, :en]}. Searched in:   *
"/Users/ryan/Sites/side_projects/blorgh/test/dummy/app/views"   *
"/Users/ryan/Sites/side_projects/blorgh/app/views"
```

The engine is unable to find the partial required for rendering the comments. Rails looks first in the application's (`test/dummy`) `app/views` directory and then in the engine's `app/views` directory. When it can't find it, it will throw this error. The engine knows to look for `blorgh/comments/_comment` because the model object it is receiving is from the `Blorgh::Comment` class.

This partial will be responsible for rendering just the comment text, for now. Create a new file at `app/views/blorgh/comments/_comment.html.erb` and put this line inside it:

```
<%= comment_counter + 1 %>. <%= comment.text %>
```

The `comment_counter` local variable is given to us by the `<%= render @article.comments %>` call, which will define it automatically and increment the counter as it iterates through each comment. It's used in this example to display a small number next to each comment when it's created.

That completes the comment function of the blogging engine. Now it's time to use it within an application.

Hooking Into an Application

Using an engine within an application is very easy. This section covers how to mount the engine into an application and the initial setup required, as well as linking the engine to a `User` class provided by the application to provide ownership for articles and comments within the engine.

Mounting the Engine

First, the engine needs to be specified inside the application's `Gemfile`. If there isn't an application handy to test this out in, generate one using the `rails new` command outside of the engine directory like this:

```
$ rails new unicorn
```

Usually, specifying the engine inside the `Gemfile` would be done by specifying it as a normal, everyday gem.

```
gem 'devise'
```

However, because you are developing the `blorgh` engine on your local machine, you will need to specify the `:path` option in your `Gemfile`:

```
gem 'blorgh', path: 'engines/blorgh'
```

Then run `bundle` to install the gem.

As described earlier, by placing the gem in the `Gemfile` it will be loaded when Rails is loaded. It will first require `lib/blorgh.rb` from the engine, then `lib/blorgh/engine.rb`, which is the file that defines the major pieces of functionality for the engine.

To make the engine's functionality accessible from within an application, it needs to be mounted in that application's `config/routes.rb` file:

```
mount Blorgh::Engine, at: "/blog"
```

This line will mount the engine at `/blog` in the application. Making it accessible at `http://localhost:3000/blog` when the application runs with `bin/rails server`.

NOTE: Other engines, such as Devise, handle this a little differently by making you specify custom helpers (such as `devise_for`) in the routes. These helpers do exactly the same thing, mounting pieces of the engines's functionality at a pre-defined path which may be customizable.

Engine Setup

The engine contains migrations for the `blorgh_articles` and `blorgh_comments` table which need to be created in the application's database so that the engine's models can query them correctly. To copy these migrations into the application run the following command from the application's root:

```
$ bin/rails blorgh:install:migrations
```

If you have multiple engines that need migrations copied over, use `railties:install:migrations` instead:

```
$ bin/rails railties:install:migrations
```

This command, when run for the first time, will copy over all the migrations from the engine. When run the next time, it will only copy over migrations that haven't been copied over already. The first run for this command will output something such as this:

```
Copied migration [timestamp_1]_create_blorgh_articles.blorgh.rb from blorgh
Copied migration [timestamp_2]_create_blorgh_comments.blorgh.rb from blorgh
```

The first timestamp (`[timestamp_1]`) will be the current time, and the second timestamp (`[timestamp_2]`) will be the current time plus a second. The reason for this is so that the migrations for the engine are run after any existing migrations in the application.

To run these migrations within the context of the application, simply run `bin/rails db:migrate` . When accessing the engine through `http://localhost:3000/blog` , the articles will be empty. This is because the table created inside the application is different from the one created within the engine. Go ahead, play around with the newly mounted engine. You'll find that it's the same as when it was only an engine.

If you would like to run migrations only from one engine, you can do it by specifying `SCOPE` :

```
$ bin/rails db:migrate SCOPE=blorgh
```

This may be useful if you want to revert engine's migrations before removing it. To revert all migrations from blorgh engine you can run code such as:

```
$ bin/rails db:migrate SCOPE=blorgh VERSION=0
```

Using a Class Provided by the Application

Using a Model Provided by the Application

When an engine is created, it may want to use specific classes from an application to provide links between the pieces of the engine and the pieces of the application. In the case of the `blorgh` engine, making articles and comments have authors would make a lot of sense.

A typical application might have a `User` class that would be used to represent authors for an article or a comment. But there could be a case where the application calls this class something different, such as `Person` . For this reason, the engine should not hardcode associations specifically for a `User` class.

To keep it simple in this case, the application will have a class called `User` that represents the users of the application (we'll get into making this configurable further on). It can be generated using this command inside the application:

```
$ bin/rails generate model user name:string
```

The `bin/rails db:migrate` command needs to be run here to ensure that our application has the `users` table for future use.

Also, to keep it simple, the articles form will have a new text field called `author_name` , where users can elect to put their name. The engine will then take this name and either create a new `User` object from it, or find one that already has that name. The engine will then associate the article with the found or created `User` object.

First, the `author_name` text field needs to be added to the `app/views/blorgh/articles/_form.html.erb` partial inside the engine. This can be added above the `title` field with this code:

```
<div class="field">
  <%= form.label :author_name %><br>
  <%= form.text_field :author_name %>
</div>
```

Next, we need to update our `Blorgh::ArticlesController#article_params` method to permit the new form parameter:

```
def article_params
  params.require(:article).permit(:title, :text, :author_name)
end
```

The `Blorgh::Article` model should then have some code to convert the `author_name` field into an actual `User` object and associate it as that article's `author` before the article is saved. It will also need to have an `attr_accessor` set up for this field, so that the setter and getter methods are defined for it.

To do all this, you'll need to add the `attr_accessor` for `author_name`, the association for the author and the `before_validation` call into `app/models/blorgh/article.rb`. The `author` association will be hard-coded to the `User` class for the time being.

```
attr_accessor :author_name
belongs_to :author, class_name: "User"

before_validation :set_author

private
  def set_author
    self.author = User.find_or_create_by(name: author_name)
  end
```

By representing the `author` association's object with the `User` class, a link is established between the engine and the application. There needs to be a way of associating the records in the `blorgh_articles` table with the records in the `users` table. Because the association is called `author`, there should be an `author_id` column added to the `blorgh_articles` table.

To generate this new column, run this command within the engine:

```
$ bin/rails generate migration add_author_id_to_blorgh_articles author_id:integer
```

NOTE: Due to the migration's name and the column specification after it, Rails will automatically know that you want to add a column to a specific table and write that into the migration for you. You don't need to tell it any more than this.

This migration will need to be run on the application. To do that, it must first be copied using this command:

```
$ bin/rails blorgh:install:migrations
```

Notice that only *one* migration was copied over here. This is because the first two migrations were copied over the first time this command was run.

```
NOTE Migration [timestamp]_create_blorgh_articles.blorgh.rb from blorgh has been
skipped. Migration with the same name already exists.
NOTE Migration [timestamp]_create_blorgh_comments.blorgh.rb from blorgh has been
skipped. Migration with the same name already exists.
Copied migration [timestamp]_add_author_id_to_blorgh_articles.blorgh.rb from blorgh
```

Run the migration using:

```
$ bin/rails db:migrate
```

Now with all the pieces in place, an action will take place that will associate an author - represented by a record in the `users` table - with an article, represented by the `blorgh_articles` table from the engine.

Finally, the author's name should be displayed on the article's page. Add this code above the "Title" output inside `app/views/blorgh/articles/show.html.erb`:

```
<p>
  <b>Author:</b>
  <%= @article.author.name %>
</p>
```

Using a Controller Provided by the Application

Because Rails controllers generally share code for things like authentication and accessing session variables, they inherit from `ApplicationController` by default. Rails engines, however are scoped to run independently from the main application, so each engine gets a scoped `ApplicationController`. This namespace prevents code collisions, but often engine controllers need to access methods in the main application's `ApplicationController`. An easy way to provide this access is to change the engine's scoped `ApplicationController` to inherit from the main application's `ApplicationController`. For our Blorgh engine this would be done by changing `app/controllers/blorgh/application_controller.rb` to look like:

```
module Blorgh
  class ApplicationController < ::ApplicationController
  end
end
```

By default, the engine's controllers inherit from `Blorgh::ApplicationController`. So, after making this change they will have access to the main application's `ApplicationController`, as though they were part of the main application.

This change does require that the engine is run from a Rails application that has an `ApplicationController`.

Configuring an Engine

This section covers how to make the `User` class configurable, followed by general configuration tips for the engine.

Setting Configuration Settings in the Application

The next step is to make the class that represents a `User` in the application customizable for the engine. This is because that class may not always be `User`, as previously explained. To make this setting customizable, the engine will have a configuration setting called `author_class` that will be used to specify which class represents users inside the application.

To define this configuration setting, you should use a `mattr_accessor` inside the `Blorgh` module for the engine. Add this line to `lib/blorgh.rb` inside the engine:

```
mattr_accessor :author_class
```

This method works like its siblings, `attr_accessor` and `cattr_accessor`, but provides a setter and getter method on the module with the specified name. To use it, it must be referenced using `Blorgh.author_class`.

The next step is to switch the `Blorgh::Article` model over to this new setting. Change the `belongs_to` association inside this model (`app/models/blorgh/article.rb`) to this:

```
belongs_to :author, class_name: Blorgh.author_class
```

The `set_author` method in the `Blorgh::Article` model should also use this class:

```
self.author = Blorgh.author_class.constantize.find_or_create_by(name: author_name)
```

To save having to call `constantize` on the `author_class` result all the time, you could instead just override the `author_class` getter method inside the `Blorgh` module in the `lib/blorgh.rb` file to always call `constantize` on the saved value before returning the result:

```
def self.author_class
  @@author_class.constantize
end
```

This would then turn the above code for `set_author` into this:

```
self.author = Blorgh.author_class.find_or_create_by(name: author_name)
```

Resulting in something a little shorter, and more implicit in its behavior. The `author_class` method should always return a `Class` object.

Since we changed the `author_class` method to return a `Class` instead of a `String`, we must also modify our `belongs_to` definition in the `Blorgh::Article` model:

```
belongs_to :author, class_name: Blorgh.author_class.to_s
```

To set this configuration setting within the application, an initializer should be used. By using an initializer, the configuration will be set up before the application starts and calls the engine's models, which may depend on this configuration setting existing.

Create a new initializer at `config/initializers/blorgh.rb` inside the application where the `blorgh` engine is installed and put this content in it:

```
Blorgh.author_class = "User"
```

WARNING: It's very important here to use the `String` version of the class, rather than the class itself. If you were to use the class, Rails would attempt to load that class and then reference the related table. This could lead to problems if the table didn't already exist. Therefore, a `String` should be used and then converted to a class using `constantize` in the engine later on.

Go ahead and try to create a new article. You will see that it works exactly in the same way as before, except this time the engine is using the configuration setting in `config/initializers/blorgh.rb` to learn what the class is.

There are now no strict dependencies on what the class is, only what the API for the class must be. The engine simply requires this class to define a `find_or_create_by` method which returns an object of that class, to be associated with an article when it's created. This object, of course, should have some sort of identifier by which it can be referenced.

General Engine Configuration

Within an engine, there may come a time where you wish to use things such as initializers, internationalization, or other configuration options. The great news is that these things are entirely possible, because a Rails engine shares much the same functionality as a Rails application. In fact, a Rails application's functionality is actually a superset of what is provided by engines!

If you wish to use an initializer - code that should run before the engine is loaded - the place for it is the `config/initializers` folder. This directory's functionality is explained in the Initializers section of the Configuring guide, and works precisely the same way as the `config/initializers` directory inside an application. The same thing goes if you want to use a standard initializer.

For locales, simply place the locale files in the `config/locales` directory, just like you would in an application.

Testing an Engine

When an engine is generated, there is a smaller dummy application created inside it at `test/dummy`. This application is used as a mounting point for the engine, to make testing the engine extremely simple. You may extend this application by generating controllers, models, or views from within the directory, and then use those to test your engine.

The `test` directory should be treated like a typical Rails testing environment, allowing for unit, functional, and integration tests.

Functional Tests

A matter worth taking into consideration when writing functional tests is that the tests are going to be running on an application - the `test/dummy` application - rather than your engine. This is due to the setup of the testing environment; an engine needs an application as a host for testing its main functionality, especially controllers. This means that if you were to make a typical `GET` to a controller in a controller's functional test like this:

```
module Blorgh
  class FooControllerTest < ActionDispatch::IntegrationTest
    include Engine.routes.url_helpers
```

```
    def test_index
      get foos_url
      # ...
    end
  end
end
```

It may not function correctly. This is because the application doesn't know how to route these requests to the engine unless you explicitly tell it **how**. To do this, you must set the `@routes` instance variable to the engine's route set in your setup code:

```
module Blorgh
  class FooControllerTest < ActionDispatch::IntegrationTest
    include Engine.routes.url_helpers

    setup do
      @routes = Engine.routes
    end

    def test_index
      get foos_url
      # ...
    end
  end
end
```

This tells the application that you still want to perform a `GET` request to the `index` action of this controller, but you want to use the engine's route to get there, rather than the application's one.

This also ensures that the engine's URL helpers will work as expected in your tests.

Improving Engine Functionality

This section explains how to add and/or override engine MVC functionality in the main Rails application.

Overriding Models and Controllers

Engine models and controllers can be reopened by the parent application to extend or decorate them.

Overrides may be organized in a dedicated directory `app/overrides`, ignored by the autoloader, and preloaded in a `to_prepare` callback:

```
# config/application.rb
module MyApp
  class Application < Rails::Application
    # ...

    overrides = "#{Rails.root}/app/overrides"
    Rails.autoloaders.main.ignore(overrides)
```

```
    config.to_prepare do
      Dir.glob("#{overrides}/**/*_override.rb").each do |override|
        load override
      end
    end
  end
end
```

Reopening existing classes using `class_eval`

For example, in order to override the engine model

```
# Blorgh/app/models/blorgh/article.rb
module Blorgh
  class Article < ApplicationRecord
    # ...
  end
end
```

you just create a file that *reopens* that class:

```
# MyApp/app/overrides/models/blorgh/article_override.rb
Blorgh::Article.class_eval do
  # ...
end
```

It is very important that the override *reopens* the class or module. Using the `class` or `module` keywords would define them if they were not already in memory, which would be incorrect because the definition lives in the engine. Using `class_eval` as shown above ensures you are reopening.

Reopening existing classes using ActiveSupport::Concern

Using `Class#class_eval` is great for simple adjustments, but for more complex class modifications, you might want to consider using [`ActiveSupport::Concern`] (https://api.rubyonrails.org/classes/ActiveSupport/Concern.html). ActiveSupport::Concern manages load order of interlinked dependent modules and classes at run time allowing you to significantly modularize your code.

Adding `Article#time_since_created` and Overriding `Article#summary`:

```
# MyApp/app/models/blorgh/article.rb

class Blorgh::Article < ApplicationRecord
  include Blorgh::Concerns::Models::Article

  def time_since_created
    Time.current - created_at
  end

  def summary
    "#{title} - #{truncate(text)}"
```

```
  end
end
```

```
# Blorgh/app/models/blorgh/article.rb
module Blorgh
  class Article < ApplicationRecord
    include Blorgh::Concerns::Models::Article
  end
end
```

```
# Blorgh/lib/concerns/models/article.rb

module Blorgh::Concerns::Models::Article
  extend ActiveSupport::Concern

  # `included do` causes the block to be evaluated in the context
  # in which the module is included (i.e. Blorgh::Article),
  # rather than in the module itself.
  included do
    attr_accessor :author_name
    belongs_to :author, class_name: "User"

    before_validation :set_author

    private
      def set_author
        self.author = User.find_or_create_by(name: author_name)
      end
  end

  def summary
    "#{title}"
  end

  module ClassMethods
    def some_class_method
      'some class method string'
    end
  end
end
```

Autoloading and Engines

Please check the Autoloading and Reloading Constants guide for more information about autoloading and engines.

Overriding Views

When Rails looks for a view to render, it will first look in the `app/views` directory of the application. If it cannot find the view there, it will check in the `app/views` directories of all engines that have this directory.

When the application is asked to render the view for `Blorgh::ArticlesController`'s index action, it will first look for the path `app/views/blorgh/articles/index.html.erb` within the application. If it cannot find it, it will look inside the engine.

You can override this view in the application by simply creating a new file at `app/views/blorgh/articles/index.html.erb`. Then you can completely change what this view would normally output.

Try this now by creating a new file at `app/views/blorgh/articles/index.html.erb` and put this content in it:

```
<h1>Articles</h1>
<%= link_to "New Article", new_article_path %>
<% @articles.each do |article| %>
  <h2><%= article.title %></h2>
  <small>By <%= article.author %></small>
  <%= simple_format(article.text) %>
  <hr>
<% end %>
```

Routes

Routes inside an engine are isolated from the application by default. This is done by the `isolate_namespace` call inside the `Engine` class. This essentially means that the application and its engines can have identically named routes and they will not clash.

Routes inside an engine are drawn on the `Engine` class within `config/routes.rb`, like this:

```
Blorgh::Engine.routes.draw do
  resources :articles
end
```

By having isolated routes such as this, if you wish to link to an area of an engine from within an application, you will need to use the engine's routing proxy method. Calls to normal routing methods such as `articles_path` may end up going to undesired locations if both the application and the engine have such a helper defined.

For instance, the following example would go to the application's `articles_path` if that template was rendered from the application, or the engine's `articles_path` if it was rendered from the engine:

```
<%= link_to "Blog articles", articles_path %>
```

To make this route always use the engine's `articles_path` routing helper method, we must call the method on the routing proxy method that shares the same name as the engine.

```
<%= link_to "Blog articles", blorgh.articles_path %>
```

If you wish to reference the application inside the engine in a similar way, use the `main_app` helper:

```
<%= link_to "Home", main_app.root_path %>
```

If you were to use this inside an engine, it would **always** go to the application's root. If you were to leave off the `main_app` "routing proxy" method call, it could potentially go to the engine's or application's root, depending on where it was called from.

If a template rendered from within an engine attempts to use one of the application's routing helper methods, it may result in an undefined method call. If you encounter such an issue, ensure that you're not attempting to call the application's routing methods without the `main_app` prefix from within the engine.

Assets

Assets within an engine work in an identical way to a full application. Because the engine class inherits from `Rails::Engine`, the application will know to look up assets in the engine's `app/assets` and `lib/assets` directories.

Like all of the other components of an engine, the assets should be namespaced. This means that if you have an asset called `style.css`, it should be placed at `app/assets/stylesheets/[engine name]/style.css`, rather than `app/assets/stylesheets/style.css`. If this asset isn't namespaced, there is a possibility that the host application could have an asset named identically, in which case the application's asset would take precedence and the engine's one would be ignored.

Imagine that you did have an asset located at `app/assets/stylesheets/blorgh/style.css`. To include this asset inside an application, just use `stylesheet_link_tag` and reference the asset as if it were inside the engine:

```
<%= stylesheet_link_tag "blorgh/style.css" %>
```

You can also specify these assets as dependencies of other assets using Asset Pipeline require statements in processed files:

```
/*
 *= require blorgh/style
 */
```

INFO. Remember that in order to use languages like Sass or CoffeeScript, you should add the relevant library to your engine's `.gemspec`.

Separate Assets and Precompiling

There are some situations where your engine's assets are not required by the host application. For example, say that you've created an admin functionality that only exists for your engine. In this case, the host application doesn't need to require `admin.css` or `admin.js`. Only the gem's admin layout needs these assets. It doesn't make sense for the host app to include `"blorgh/admin.css"` in its stylesheets. In this situation, you should explicitly define these assets for precompilation. This tells Sprockets to add your engine assets when `bin/rails assets:precompile` is triggered.

You can define assets for precompilation in `engine.rb`:

```
initializer "blorgh.assets.precompile" do |app|
  app.config.assets.precompile += %w( admin.js admin.css )
end
```

For more information, read the Asset Pipeline guide.

Other Gem Dependencies

Gem dependencies inside an engine should be specified inside the `.gemspec` file at the root of the engine. The reason is that the engine may be installed as a gem. If dependencies were to be specified inside the `Gemfile`, these would not be recognized by a traditional gem install and so they would not be installed, causing the engine to malfunction.

To specify a dependency that should be installed with the engine during a traditional `gem install`, specify it inside the `Gem::Specification` block inside the `.gemspec` file in the engine:

```
s.add_dependency "moo"
```

To specify a dependency that should only be installed as a development dependency of the application, specify it like this:

```
s.add_development_dependency "moo"
```

Both kinds of dependencies will be installed when `bundle install` is run inside of the application. The development dependencies for the gem will only be used when the development and tests for the engine are running.

Note that if you want to immediately require dependencies when the engine is required, you should require them before the engine's initialization. For example:

```
require "other_engine/engine"
require "yet_another_engine/engine"

module MyEngine
  class Engine < ::Rails::Engine
  end
end
```

Load and Configuration Hooks

Rails code can often be referenced on load of an application. Rails is responsible for the load order of these frameworks, so when you load frameworks, such as `ActiveRecord::Base`, prematurely you are violating an implicit contract your application has with Rails. Moreover, by loading code such as `ActiveRecord::Base` on boot of your application you are loading entire frameworks which may slow down your boot time and could cause conflicts with load order and boot of your application.

Load and configuration hooks are the API that allow you to hook into this initialization process without violating the load contract with Rails. This will also mitigate boot performance degradation and avoid conflicts.

Avoid loading Rails Frameworks

Since Ruby is a dynamic language, some code will cause different Rails frameworks to load. Take this snippet for instance:

```
ActiveRecord::Base.include(MyActiveRecordHelper)
```

This snippet means that when this file is loaded, it will encounter `ActiveRecord::Base`. This encounter causes Ruby to look for the definition of that constant and will require it. This causes the entire Active Record framework to be loaded on boot.

`ActiveSupport.on_load` is a mechanism that can be used to defer the loading of code until it is actually needed. The snippet above can be changed to:

```
ActiveSupport.on_load(:active_record) do
  include MyActiveRecordHelper
end
```

This new snippet will only include `MyActiveRecordHelper` when `ActiveRecord::Base` is loaded.

When are Hooks called?

In the Rails framework these hooks are called when a specific library is loaded. For example, when `ActionController::Base` is loaded, the `:action_controller_base` hook is called. This means that all `ActiveSupport.on_load` calls with `:action_controller_base` hooks will be called in the context of `ActionController::Base` (that means `self` will be an `ActionController::Base`).

Modifying Code to use Load Hooks

Modifying code is generally straightforward. If you have a line of code that refers to a Rails framework such as `ActiveRecord::Base` you can wrap that code in a load hook.

Modifying calls to `include`

```
ActiveRecord::Base.include(MyActiveRecordHelper)
```

becomes

```
ActiveSupport.on_load(:active_record) do
  # self refers to ActiveRecord::Base here,
  # so we can call .include
  include MyActiveRecordHelper
end
```

Modifying calls to `prepend`

```
ActionController::Base.prepend(MyActionControllerHelper)
```

becomes

```
ActiveSupport.on_load(:action_controller_base) do
  # self refers to ActionController::Base here,
  # so we can call .prepend
  prepend MyActionControllerHelper
end
```

Modifying calls to class methods

```
ActiveRecord::Base.include_root_in_json = true
```

becomes

```
ActiveSupport.on_load(:active_record) do
  # self refers to ActiveRecord::Base here
  self.include_root_in_json = true
end
```

Available Load Hooks

These are the load hooks you can use in your own code. To hook into the initialization process of one of the following classes use the available hook.

Class	Hook
ActionCable	action_cable
ActionCable::Channel::Base	action_cable_channel
ActionCable::Connection::Base	action_cable_connection
ActionCable::Connection::TestCase	action_cable_connection_test_case
ActionController::API	action_controller_api
ActionController::API	action_controller
ActionController::Base	action_controller_base
ActionController::Base	action_controller
ActionController::TestCase	action_controller_test_case
ActionDispatch::IntegrationTest	action_dispatch_integration_test
ActionDispatch::Response	action_dispatch_response
ActionDispatch::Request	action_dispatch_request
ActionDispatch::SystemTestCase	action_dispatch_system_test_case
ActionMailbox::Base	action_mailbox
ActionMailbox::InboundEmail	action_mailbox_inbound_email
ActionMailbox::Record	action_mailbox_record
ActionMailbox::TestCase	action_mailbox_test_case
ActionMailer::Base	action_mailer
ActionMailer::TestCase	action_mailer_test_case
ActionText::Content	action_text_content

ActionText::Record	action_text_record
ActionText::RichText	action_text_rich_text
ActionView::Base	action_view
ActionView::TestCase	action_view_test_case
ActiveJob::Base	active_job
ActiveJob::TestCase	active_job_test_case
ActiveRecord::Base	active_record
ActiveStorage::Attachment	active_storage_attachment
ActiveStorage::VariantRecord	active_storage_variant_record
ActiveStorage::Blob	active_storage_blob
ActiveStorage::Record	active_storage_record
ActiveSupport::TestCase	active_support_test_case
i18n	i18n

Available Configuration Hooks

Configuration hooks do not hook into any particular framework, but instead they run in context of the entire application.

Hook	Use Case
before_configuration	First configurable block to run. Called before any initializers are run.
before_initialize	Second configurable block to run. Called before frameworks initialize.
before_eager_load	Third configurable block to run. Does not run if config.eager_load set to false.
after_initialize	Last configurable block to run. Called after frameworks initialize.

Configuration hooks can be called in the Engine class.

```
module Blorgh
  class Engine < ::Rails::Engine
    config.before_configuration do
      puts 'I am called before any initializers'
    end
  end
end
```