Skip to content

RailsApps / rails_apps_composer

Search or jump to...

- In this repository All GitHub ↵
  Jump to ↵

- No suggested jump to results

- Octocat Spinner Icon

- Dashboard
- Pull requests
- Issues
- Marketplace
- Explore

-
- ▶ +
- k-takami
  ▶

Sign out

Sign out

- Watch 60
  ✕ Notifications
  ✓
  [✓ Not watching] Be notified when participating or @mentioned. Watch
  ✓
  Watching Be notified of all conversations. Unwatch
  ✓
  Ignoring Never be notified. Stop ignoring
- Unstar 1,430
  Star 1,430
- Fork
  315

RailsApps/rails_apps_composer

tutorial rails apps composer

Derek Kniffin edited this page Dec 29, 2017 · 9 revisions

Guide to the Rails Apps Composer Gem

by Daniel Kehoe

The Rails Apps Composer gem installs a command line tool to assemble a Rails application from a collection of “recipes.”

What you can do with the rails_apps_composer gem:

- create a starter app from the command line
- make a reusable application template

The rails_apps_composer gem is used to build the example applications and tutorials from the RailsApps project.

Convenient links:

- GitHub repository for rails_apps_composer
- RubyGems page for rails_apps_composer
- The Ruby Toolbox listing for rails_apps_composer

rails_apps_composer

A gem with recipes to create Rails application templates you can use to generate Rails starter apps.

INSTALL >
```
gem install
rails_apps_composer
```

Authors
Daniel Kehoe

16,494 total downloads
377 for this version

Owners

Links Homepage

Follow on Twitter

Follow the project on Twitter: @rails_apps. Tweet some praise if you like what you've found.

Who This Is For

This gem is for experienced Rails developers who need a tool to create and maintain a Rails starter app or Rails application template. If you're regularly building Rails applications that use certain popular gems or rely on common functionality such as authentication, you can benefit from the rails_apps_composer gem.

If you are new to Rails, try the Rails Composer tool instead. It provides an application template that is generated from the rails_apps_composer gem's core recipes. It is easier to use.

Table of Contents

- Overview
- Recipes
- Usage
- Diagnostics
- Hacking the Gem
- Anatomy of a Recipe
- Architecture
- Defaults File

- Issues

Introduction

The rails_apps_composer gem was originally developed as a tool for the RailsApps project. The RailsApps project provides Rails example applications and tutorials.

Over a hundred Rails developers now use the tool to create their own starter apps. An active open source community contributes to maintenance and development of the gem with patches and new recipes.

The rails_apps_composer gem serves as a common point of integration for many popular Rails gems. As you are no doubt aware, Rails is not just a Ruby gem, it is a complex and rapidly evolving ecosystem. Not all gems work together; sometimes new gem versions introduce unexpected incompatibilities. By using the rails_apps_composer gem to build your Rails starter apps, you will benefit from the collaborative problem-solving of Rails developers who face the same challenges as you. Check the GitHub issues if you find problems; share solutions by submitting pull requests.

Alternatives

The rails_apps_composer gem is derived from Michael Bleigh's RailsWizard gem. Early versions of the rails_apps_composer gem were a close fork of Michael Bleigh's gem; rails_apps_composer 2.0 differs substantially. Other notable alternatives are Dr. Nic Williams's App Scrolls and Daniel Davey's app_drone. See a list of Rails Application Template Projects, particularly a list of "Application template-generating gems" for similar projects.

Dependencies

Before using the rails_apps_composer gem, you will need:

- The Ruby language (version 1.9.3)
- Rails 3.2 or newer

See the article Installing Rails for advice about updating Rails and your development environment. You'll avoid many potential headaches if you review the article step by step before beginning development.

Installation

I recommend installing and using rvm, the Ruby Version Manager, to create a new gemset for rails_apps_composer. Using an rvm gemset will make it easier to identify and isolate incompatibilities among gems.

Installation is simple:

```
$ gem install rails_apps_composer
```

Overview

This guide covers use of the rails_apps_composer gem as:

- a tool to build a **starter app** from recipes
- a tool to generate a reusable **application template**

There are two ways to use rails_apps_composer as a tool:

- interactively, driven from prompts on the command line
- “automatically,” from a defaults file that specifies parameters

This guide shows how to use the recipes that come with the gem.

You can also use **local recipes** you’ve created for yourself.

This guide shows how to **modify and create recipes** to accommodate your specific development needs. I encourage you to contribute your recipes to be included in the gem for others to use.

If you intend to use the rails_apps_composer gem frequently, I recommend creating a **defaults file** that drives the assembly of your starter apps. The defaults file will eliminate the need to respond to prompts when you build a starter app interactively or generate an application template. Note that you can both set defaults and allow prompts for options which vary with each build.

Here are some of the ways you might use the rails_apps_composer gem.

Use Case: One-Off Applications for a Solo User

If you build Rails applications on your own, for example, as a hobbyist or solo consultant, you’ll likely use rails_apps_composer interactively, creating a new Rails application each time you start a new development project. For this use case, you may be better served by using the Rails Composer tool which is based on a generic application template generated by rails_apps_composer gem. You’ll only need to use the rails_apps_composer gem if the Rails Composer tool doesn’t provide the options you need for your own projects. You’ll find the rails_apps_composer gem is particularly useful if you are an infrequent Rails developer (for example, if you oscillate between Rails development and other consulting services) because the open source community will keep your starter app updated while you are away from Rails.

Use Case: Application Templates for Consulting Firms

If you are part of a team that creates new Rails applications frequently, you'll likely use the rails_apps_composer gem to generate an application template. This is a common scenario in a consulting firm or development shop where the same starter app is used for many client projects. Your team may already have an application template or a code repository for a starter app; you might want to base your starter app on the rails_apps_composer gem so you can benefit from the shared problem-solving of the open source community. You'll generate an application template using the rails_apps_composer gem so everyone on your team can get the same starter app with a single command such as `rails new -m team_starter_app.rb`.

Use Case: Example Applications for Open Source Projects

If you maintain an open source project such as a Rails gem, you may want to offer an example application that shows how your gem is used in an application. An example application is a powerful aid in encouraging adoption of your gem and providing documentation and support. It can be difficult to continue updating an example application as Rails (and its dependencies) evolves. Use the rails_apps_composer gem to create an example application that uses your gem and it will be easier to keep the example current. As an additional benefit, you can provide an application template on your website to make it easy for users to install the example application.

Terminology

If you are new to Rails, you need to know that a Ruby **gem** is a Ruby program or library distributed in self-contained format that can be installed using the RubyGems package manager. Ruby gems extend the functionality of your Rails application. Most developers begin development of any Rails application by specifying all the gems they need in the Gemfile. Some gems are ready to use as soon as they are added to the Gemfile (and the developer runs `bundle install`); other gems require the developer to run a **generator** to create configuration files or prepare the application. The rails_apps_composer gem is not added to the application Gemfile; it is a tooling gem that adds to the functionality of your system Ruby installation. It installs the gems that most developers need. And it runs the generators needed to set up the gems.

Application template has a specific meaning among Rails developers. It is a Ruby program that uses methods provided by the Rails::Generators::Actions module that powers the `rails new` command. An application template offers a mechanism to customize a Rails application immediately after a new Rails application is created with the `rails new` command. The term "template" should not be confused with the Rails "view templating" system that embeds Ruby into a text document for rendering as HTML by an ActionView::TemplateHandler such as ERB or Haml. A Rails application template is a script which programmatically modifies a new Rails application. In this respect, a Rails application template is very different from an "HTML template" or a "Word document template" which are simply sample files, The rails_apps_composer gem assembles an application template from **recipes** (the term originated with Michael Bleigh's RailsWizard gem). A recipe is snippet of Ruby code that manipulates files to set up a Rails application. A recipe is largely procedural code (a series of steps with some conditional execution). A recipe is not a self-contained program; recipes must be assembled into a script by the rails_apps_composer program. The recipes have a special format that makes it possible to set configuration prompts and run-time parameters within the recipe file. Internally, recipes are saved as objects in the `recipes[]` array so any recipe can know what other recipes are in use.

Recipe **configuration parameters** and global **preferences** are used to express your preferences for choices offered by a recipe. For example, a recipe might ask you to choose between SQLite and PostgreSQL for your database. Each recipe has access to configuration parameters in a `config{}` hash that is local to the recipe. These configuration parameters can be set with an interactive prompt. Each recipe also has access to preferences which are made available to all the recipes via the `prefs{}` hash.

Configuration parameters can be set at an interactive prompt (the rails_apps_composer program will "quiz" you for your preferences) or they can be specified in a **defaults file** so the program will run automatically with prior knowledge of your preferences. You can have more than one defaults file; for example, if you want different starter apps for different types of web applications. Parameters in the defaults file are saved internally as:

- `recipes[]` array
- `prefs{}` hash
- `gems[]` array

Your **starter app** is the "template" (there's that ambiguous term again!) you'll use as the foundation for the custom application you build for yourself or your clients. Rails gives you a simple application in a standard configuration when you use the `rails new` command; it is seldom everything you need to get started. That's why we build starter apps.

So now you understand: When you use the rails_apps_composer gem, you'll save *configuration parameters* in a *defaults file* which set up *recipes* that are assembled by the rails_apps_composer gem to generate an *application template* which is used by the `rails new` command to build a Rails *starter app* which you'll customize to deploy as a Rails application.

First Encounter

Here's what you can try on a first encounter with the rails_apps_composer gem.

First, examine the recipes by looking at the files in the recipes folder. Get a sense of how the recipes work and identify which recipes are useful or might be adapted for your own needs.

Install the rails_apps_composer gem and try it from the command line.

Try `rails_apps_composer list` to see a list of recipes. Try `rails_apps_composer new myapp` to see how the gem prompts for recipes and extra gems and then generates an application. Select the *core* recipe for a sampling of the program's options.

Make a list of recipes and try generating an application by providing your list of recipes on the command line. Here's a simple example: `rails_apps_composer new myapp -r setup readme gems extras`.

Try generating an application template using recipes: `rails_apps_composer template ~/Desktop/template.rb -r setup readme gems extras`.

Create an application from the application template using the command `rails new myapp -m ~/Desktop/template.rb`.

Each command is explained below in the Usage section.

Recipes

You must examine the source code for the recipes to make full use of the rails_apps_composer gem. See the repository recipes directory. The recipes are regularly updated and contributors add new recipes frequently. Not all recipes are described here.

Local Recipes

If you need recipes that are only useful for your own projects, create recipes (see Anatomy of a Recipe) and the rails_apps_composer program can use them as well as the recipes that are built in to the gem.

Collections

A recipe can require other recipes, so it's possible to create a recipe that loads a collection of other recipes.

The *core* recipe simply loads all the core recipes. I recommend selecting the *core* recipe to become familiar with the program's capabilities.

Core Recipes

The following recipes were released with rails_apps_composer 2.0. These recipes are assembled to create the example applications released by the RailsApps project The core recipes are known to work together without incompatibilities.

Git

The *git* recipe initializes a git repository for your application and makes a first commit. This recipe should precede all others (put it first in any list). If the *git* recipe is included, all other recipes will make incremental commits to a git repository as the program runs.

RailsApps

The *railsapps* recipe installs any of the RailsApps example applications. Selecting an example application sets preferences that create an application identical to the selected example application. There is a tutorial available for each of the example applications; see the RailsApps project page.

Setup

The *setup* recipe is used to set configuration parameters which will be used by other recipes. Any individual recipe can prompt for configuration parameters that will be consumed only by that recipe. You'll use the *setup* recipe to set configuration parameters that need to be known by more than one recipe (for example, whether you prefer ERB or Haml for your templating engine). Use the *setup* recipe to set configuration parameters for all the recipes; these configuration parameters are be saved as key/value pairs in the `prefs{}` hash.

Readme

It's important to create a README file; often it is neglected when a developer creates an application. The *readme* recipe generates a README file that shows the Ruby and Rails versions and other choices made by a developer in creating the application. These details are very helpful when other developers look at the application. You can customize the *readme* recipe if you want your own information included in the README files.

If you include the *readme* recipe when you generate an application, the generated README file will contain a diagnostics report (unless the program aborts before the readme recipe runs). Please include the diagnostics report when you create an issue on GitHub.

Gems

The *gems* recipe adds required gems to the Gemfile and runs any generators required by the gems. If you add a custom recipe, you have a choice of adding the gem in your custom recipe or adding it to the *gems* recipe. As a rule of thumb, if the gem you want to use is popular and a common requirement that any developer will want, add it to the *gems* recipe. If the gem is of limited usefulness, add it to your own custom recipe and let other developers add your custom recipe optionally.

You don’t have to write a recipe to add a gem. Write custom recipes or modify the *gems* recipe only if you are adding a gem that requires use of a generator or manipulation of your starter app’s files. If you only need to add gems to the Gemfile, the rails_apps_composer program will prompt you to add gems when you run it interactively. You can also specify gems that you wish to add to the Gemfile in a rails_apps_composer defaults file. This eliminates the need to write custom recipes or modify the *gems* recipe if all you need to do is add gems to a Gemfile.

Testing

The *testing* recipe allows you to specify your testing framework. The core recipe offers a choice of popular frameworks for unit testing (Test::Unit or RSpec) and integration testing (Cucumber) as well as fixture replacements (Factory Girl or Machinist). If you prefer a testing framework that is not offered in the core recipe, please add it to the *testing* recipe and submit a pull request so we can make your choice available to others.

The *testing* recipe is one of the more complex recipes because setting up a testing framework often requires running generators and modifying configuration files based on other choices you’ve made (such as a choice of database or authentication).

Email

Many applications must send email, especially “transactional email” (for example, when an authentication system such as Devise requires confirmation of an email address). The *email* recipe configures ActionMailer for various email accounts, including Gmail, SMTP, SendGrid, and Mandrill. If you prefer a different email provider, add it to the *email* recipe and submit a pull request.

Models, Controllers, Views, Routes

Select the *models*, *controllers*, *views*, and *routes* recipes to create simple starter apps for your own project. Depending on other preferences you’ve selected, you can choose:

```
question  Install a starter app?
      1)  None
      2)  Home Page
      3)  Home Page, User Accounts
      4)  Home Page, User Accounts, Admin Dashboard
      5)  Home Page, User Accounts, Subdomains
```

Rather than installing gems or setting configuration parameters, these recipes install the “internal organs” of entire simple applications.

The simplest use of these four recipes is to create a home page.

If you’re creating your own custom starter app, you can examine these recipes for implementation ideas; you may create your own custom recipes that are similar to these recipes. These four recipes are closely related and could have been combined into one "mvc" recipe except the resulting file would have been large and cumbersome.

The *models* recipe creates a User model. The *models* recipe sets up both authentication and authorization for your starter app. The core recipe accommodates a choice of Devise or OmniAuth for authentication and CanCan for authorization. Authentication and authorization require complex changes to the basic Rails application so this recipe is more complex than others.

The *controllers* recipe is relatively simple. Instead of generating controllers using the `rails generate` command, we download and install controller files from the RailsApps example application GitHub repositories. Where conditions require it (for example, injecting methods for managing authorization), we manipulate the files using the rails_apps_composer `gsub_file` method (for regex search and replace).

The *views* recipe also downloads files from the RailsApps repos. The rails_apps_composer program supplies a special `copy_from_repo` method that automatically translates ERB view templates into Haml or Slim code if you've stated you prefer Haml or Slim.

The *routes* recipe adds the routes needed for the RailsApps example applications. For a simple home page, the standard **config/routes.rb** file is manipulated with the `gsub_file` method. For more complex applications, routes files are downloaded from the RailsApps example apps. Note that downloaded files must be tweaked with `gsub_file` to set the correct application name.

Frontend

The *frontend* recipe installs a default application layout file, complete with partials for navigation links and Rails flash messages. The files are downloaded from the RailsApps repos. The rails_apps_composer `copy_from_repo` method automatically translates ERB view templates into Haml or Slim code as needed. The *frontend* recipe also installs CSS and Javascript files, adding Twitter Bootstrap or other front-end frameworks as required.

You can modify the *frontend* recipe (and submit a pull request) if you have a favorite front-end configuration that other developers might like. For example, rails_apps_composer 2.0 didn't include Thoughtbot's popular Bourbon Sass mixin library; you could modify the *frontend* recipe to offer it as an option.

Init

Use the *init* recipe to initialize a database using a **db/seeds.rb** file and run any migrations needed by your SQL database. The *init* recipe is important for the RailsApps example applications as it makes the example apps ready to run with sample data. You can examine the recipe for implementation ideas; you might add similar code to your own custom recipes.

Extras

The *extras* recipe is a grab bag of ingredients. In rails_apps_composer 2.1, the *extras* recipe offers:

- an option to install a form builder gem
- a “ban_spiders” option to control search-engine spidering by modifying the **public/robots.txt** file
- a “jsruntime” option to add ‘therubyracer’ JavaScript runtime for Linux users without node.js
- an “rvmrc” option to create a project-specific rvm gemset
- a housekeeping procedure to remove unnecessary files and whitespace
- an option to create a GitHub repository

Each of these procedures could be moved into its own recipe. However, since the procedures only require a few lines of code and will be useful to a large number of developers, we’ve consolidated them in the *extras* recipe.

If you have a favorite option that you think many other developers would like, feel free to customize the *extras* recipe and submit a pull request. Keep in mind that every developer who builds a starter app with the rails_apps_composer gem will have to respond to prompts introduced by the *extras* recipe. If you’re not certain that everyone will want to consider your favorite option, create a custom recipe instead and allow developers a choice of including it in their recipe list.

Additional Recipes

The rails_apps_composer gem does not offer every option that a Rails developer will want in creating a starter app. That’s why it’s built using recipes. It’s an open source project so it’s up to you to contribute recipes that add functionality you need and other developers may want. Look at the rails_apps_composer source code in the recipes folder. Maybe someone has already contributed a recipe you can use. Otherwise, write a recipe and submit a pull request so we can add it to the project.

See the section below, Hacking the Gem, to learn how to contribute new recipes.

Usage

These commands are summarized in the project README file for your reference.

List Recipes

You can list recipes with short descriptions:

```
$ rails_apps_composer list

controllers   # Add controllers needed for starter apps.
core          # Select all core recipes.
email         # Configure email accounts.
example       # Example of a recipe.
extras        # Various extras.
frontend      # Install a front-end framework for HTML5 and CSS.
gems          # Add the gems your application needs.
git           # Initialize git for your application.
init          # Set up and initialize database.
models        # Add models needed for starter apps.
railsapps     # Install RailsApps example applications.
readme        # Build a README file for your application.
routes        # Add routes needed for starter apps.
setup         # Make choices for your application.
testing       # Add testing framework.
views         # Add views needed for starter apps.
```

The above list is current for rails_apps_composer 2.1; later releases will contain more recipes.

You can find all available recipes in the repository recipes directory. Examining the recipe source code is the very best way to learn what a recipe will do.

Recommended Recipes

I recommend selecting the *core* recipe if you are just getting started. The *core* recipe installs a collection of the most useful recipes.

Recipe Order and Interdependency

The order in which you input a list or recipes determines the order of execution unless a recipe contains a `requires` or `run_after` directive.

A recipe may contain a `requires` directive which specifies other recipes which must be present before a recipe can run. The `requires` constraint will force the rails_apps_composer program to load any required recipes, even if you don't add them explicitly.

Recipes may also contain a `run_after` directive. The rails_apps_composer program will organize the order of execution so that any recipes in the `run_after` list will execute earlier. That is, the recipe will run after everything else in the list.

In general, it's best to add (or list) recipes in the order they should execute as some recipes may not contain a necessary `requires` or `run_after` directive.

See the Anatomy of a Recipe section in the Guide to learn about the format of a recipe.

Skipping Test::Unit or Active Record

If you plan to use RSpec instead of Test::Unit, or use an ORM such as Mongoid instead of Active Record, you must pass the `-T` or `-O` flags to the program so it will skip Test::Unit or Active Record.

The rails_apps_composer program will ask if you want to skip Test::Unit or Active Record when you generate an application interactively.

Any recipe can set the `-T` or `-O` arguments using the `args` directive.

Also, you can set `-T` or `-O` arguments in the defaults file.

Setting `-T` or `-O` arguments has no affect on application templates. An application template runs after the `rails new` command so it cannot set the `-T` or `-O` arguments. When you generate an application template, the program will not ask if you want to skip Test::Unit or Active Record.

Using Local Recipes

You can use local recipes you've created yourself by using the `-l` flag and supplying the name of a directory that contains local recipes. The `-l` flag can be combined with any other command so you can create applications or application templates, interactively or from a defaults file.

For example, generate an application interactively using local recipes:

```
$ rails_apps_composer new myapp -l ~/recipes/
```

If you create local recipes, please consider contributing them to the project.

Generate an Application Interactively

You'll be prompted for recipes and gems:

```
$ rails_apps_composer new myapp

Would you like to skip Test::Unit? (yes for RSpec) (y/n)
Would you like to skip Active Record? (yes for MongoDB) (y/n)

Available Recipes:
collections: core
configuration: email, gems, git, railsapps, readme, setup
example: example
frontend: frontend
initialize: init
mvc: controllers, models, routes, views
other: extras
testing: testing

Which recipe would you like to add? (blank to finish)
```

```
What gem would you like to add? (blank to finish)

Generating basic application, using:
"rails new myapp -m <temp_file>"
```

If you want to skip the prompts asking about Test::Unit and Active Record, you can set these arguments in a defaults file.

You will be able to choose recipes that are contained in the rails_apps_composer gem plus any local recipes you've added with the `-l` argument.

Choose the *core* recipe if you are not sure which recipes to select.

You can specify any gem. Any string you enter will be added as a gem in the starter app Gemfile.

Generate an Application from a List of Recipes

Provide a list of recipes using the `-r` flag. In this example, the *core* recipe selects all available core recipes:

```
$ rails_apps_composer new myapp -r core

Would you like to skip Test::Unit? (yes for RSpec) (y/n)
Would you like to skip Active Record? (yes for MongoDB) (y/n)
What gem would you like to add? (blank to finish)

Generating basic application, using:
"rails new myapp -m <temp_file>"
```

The program will prompt you for your preferences before generating an application.

Generate an Application from Defaults

Use a defaults file for recipes, preferences, and extra gems so you don't have to provide them interactively. Then generate an application using the `-d` flag:

```
$ rails_apps_composer new myapp -d my_defaults.yaml
```

Use the `my_defaults.yaml` file to specify a list of recipes, preferences, and extra gems. You can use any name (and file extension) for the file. See the Defaults File section in the Guide concerning the format of the defaults file.

Generate an Application Template Interactively

You may want an application template to share with others. For an example, see the Rails Composer project which is an application template generated from the rails_apps_composer core recipes.

Specify a filename for the template:

```
$ rails_apps_composer template ~/Desktop/template.rb

Available Recipes:
collections: core
configuration: email, gems, git, railsapps, readme, setup
example: example
frontend: frontend
initialize: init
mvc: controllers, models, routes, views
other: extras
testing: testing

Which recipe would you like to add? (blank to finish)
What gem would you like to add? (blank to finish)
Generating and saving application template...
Done.
```

The command with the `template` argument followed by a filename generates an application template. You can add additional recipes or gems when prompted.

Generate an Application Template from a List of Recipes

Specify a filename for the template and provide a list of recipes:

```
$ rails_apps_composer template ~/Desktop/template.rb -r core

What gem would you like to add? (blank to finish)
Generating and saving application template...
Done.
```

The command with the `template` argument followed by a filename and a list of recipes generates an application template.

Generate an Application Template from Defaults

Generate an application template using a defaults file and the `-d` flag:

```
$ rails_apps_composer template ~/Desktop/template.rb -d my_defaults.yaml
```

Use the `my_defaults.yaml` file to specify a list of recipes, preferences, and extra gems. You can use any name (and file extension) for the file. See the Defaults File in the Guide concerning the format of the defaults file.

Generate an Application from a Template

After you've created a template file, you can generate an application from a template at any time using the `rails new` command with the `-m` option:

```
$ rails new myapp -m ~/Desktop/template.rb
```

The application template will prompt you for any configuration preferences requested by the recipes.

The `rails new` command allows you to specify the `-T -O` flags to skip Test::Unit files and Active Record files when you use an application template.

```
$ rails new myapp -m ~/Desktop/template.rb -T -O
```

The `rails new` command with the `-m` option also allows you to specify an application template that can be downloaded via HTTP. This makes it possible to host a template on GitHub (for example) and allow anyone to generate an application from the hosted template. For example:

```
$ rails new myapp -m https://raw.github.com/RailsApps/rails-composer/master/composer.rb
```

Diagnostics

The rails_apps_composer program runs diagnostics to reveal whether you've picked recipes and preferences that are known to work together.

Hundreds of developers are using this gem to build starter apps. As you can see from the commit log, the gem is actively maintained and the collection of recipes is growing. There is a drawback to all this activity, however. No developer who uses the rails_apps_composer gem uses all the recipes. And it would be very difficult to test a recipe in every possible combination. Consequently, combining some recipes or preferences may not work.

The rails_apps_composer gem contains an internal database of combinations of recipes and preferences that are known to work together. If you've picked recipes and preferences that are known to work together, you'll get a confirmation:

```
WOOT! The recipes you've selected are known to work together.
If they don't, open an issue for rails_apps_composer on GitHub.
WOOT! The preferences you've selected are known to work together.
If they don't, open an issue for rails_apps_composer on GitHub.
```

You'll get a warning if you've picked a combination we don't know about:

```
WARNING! The recipes you've selected might not work together.
Help us out by reporting whether this combination works or fails.
WARNING! The preferences you've selected might not work together.
Help us out by reporting whether this combination works or fails.
```

Go to the GitHub repository to create an issue and let us know if you've found a combination of recipes or preferences that work together. If you've found a combination that works together, we'll update the gem's internal database for the next public release.

It's a good idea to include the "readme" recipe when you run rails_apps_composer. If you do, the application's **README** file will contain a diagnostics report. Here's an example of the diagnostics report:

```
Recipes:
["gems", "readme", "setup"]

Preferences:
{:git=>true, :dev_webserver=>"webrick", :database=>"sqlite", :templates=>"erb", :email=>"none"}
```

Please include the diagnostics report when you create an issue on GitHub.

Hacking the Gem

This guide explains how to customize the gem either for your own use or to share with others.

You'll need to fork the gem, learn more about the anatomy of the recipes, make changes, build and install the gem locally, and submit a pull request if you've made changes you wish to share.

Gem Development Dependencies

Here's where it is advisable to use rvm, the Ruby Version Manager, to create a project-specific gemset for rails_apps_composer. Using an rvm gemset will make it easier to isolate the gems used in developing rails_apps_composer from other gems on your system.

```
$ rvm use ruby-1.9.3@rails_apps_composer --create
```

Install the gems needed for development:

```
$ gem install rails i18n thor mg rspec
```

Now you can download the source code and begin hacking.

Forking the Gem

If you wish to add or customize rails_apps_composer recipes, please consider forking the repository on GitHub. Your changes may be useful to others.

GitHub provides simple, clear instructions about forking a repo and contributing changes to a project.

Here's a summary.

Visit the rails_apps_composer repository and click "Fork".

Clone your fork:

```
$ git clone https://github.com/username/rails_apps_composer.git
```

Track the upstream repository:

```
$ cd rails_apps_composer
$ git remote add upstream https://github.com/RailsApps/rails_apps_composer.git
```

The section below, “Building the Gem,” explains how to build the gem after making changes.

Rake Tasks

After you’ve cloned the source code to your local machine, take a look at the rake tasks that are available:

```
$ rake -T

rake clean          # Remove any temporary products.
rake clobber        # Remove any generated file.
rake gem            # Build gem into dist/
rake gem:install    # Build and install as local gem
rake gem:publish    # Push the gem to RubyGems.org
rake package        # Build gem and tarball into dist/
rake print          # Prints out a template from the provided recipes.
rake reinstall      # uninstall rails_apps_composer gem and install a new version
rake run            # Execute a test run with the specified recipes.
rake spec           # run specs
```

You’ll use some of these tasks to build and install the gem.

Building the Gem

After you’ve forked the gem and cloned it locally, see the section below, Anatomy of a Recipe, to learn how to construct recipes.

After you’ve made changes, you’ll build and install the gem locally.

Each time you build and install the gem, you’ll have a distribution version saved locally in the gem’s **dist/** directory. Remove old versions:

```
$ rm dist/rails_apps_composer-*.gem
```

Remove any older versions of the installed gem from your system:

```
$ gem uninstall rails_apps_composer -x
```

Run tests:

```
$ rake spec
```

The tests can’t determine if you’ve made significant errors in your recipes. The tests only serve to check that basic functions of the gem behave as expected.

Use a rake task to build the gem:

```
$ rake gem
```

Use a rake task to install the gem locally:

```
$ rake gem:install
```

These four steps are combined in a single rake task:

```
$ rake reinstall
```

You can use either the individual steps or the combined rake task.

Pull Requests and Public Releases

After you make your changes and test your gem, submit a pull request so I can incorporate your changes in the next public release of the gem. To submit a pull request, visit the GitHub repo for your version of the rails_apps_composer gem. Click “Pull Request” and provide a description of your changes. In general, I try to respond to pull requests within a few days.

I seldom publish a new public release of the gem immediately after responding to a pull request. Typically I wait a few weeks to collect several contributions before making a new release.

This is the rake task that I use to push a new release to the RubyGems server:

```
$ rake gem:publish
```

It won’t work for you. The RubyGems server will only respond when an authorized project maintainer pushes a new release.

When the new version of the gem is published, everyone can use the new changes.

To be notified of new releases, you can visit the RubyGems page for rails_apps_composer and subscribe to the gem to be notified via an RSS feed. Alternatively, you can watch the project on GitHub or follow @rails_apps on Twitter.

When a new version is released, uninstall the local version of the gem and install the published gem:

```
$ gem uninstall rails_apps_composer -x
$ gem install rails_apps_composer
```

Now the newest public release of the gem is available for your use.

Anatomy of a Recipe

If you wish to modify or create recipes, you should examine the existing recipes.

This section shows the structure of a recipe and describes the methods provided by the rails_apps_composer program. Learn about methods provided by the rails_apps_composer program to better understand the recipes.

Recipe Structure

A recipe is a snippet of Ruby code that can be assembled into an application template by the rails_apps_composer program. It has a specific structure and uses a small set of methods provided by the rails_apps_composer program.

Here is a simple recipe. It prompts for a preference, installs a gem, and then creates a model, controller, and view.

```
before_config do
  # Code here is run before any configuration prompts.
end

# read recipe configuration parameters and set global preferences
case config['foo']
  when 'foobar'
    prefs[:foo] = 'foobar'
  when 'foobaz'
    prefs[:foo] = 'foobaz'
end

# add a gem to the Gemfile
gem 'foobar' if prefer :foo, 'foobar'
gem 'foobaz' if prefer :foo, 'foobaz'

stage_two do
  # Code here is run after Bundler installs all the gems for the project.
  # Use this section to run generators and rake tasks.
  # Download any files from a repository for models, controllers, views, and routes.
  copy_from_repo 'app/models/foo.rb'
  copy_from_repo 'app/controllers/foo_controller.rb'
  copy_from_repo 'app/views/foo/index.html.erb'
  route "root :to => 'foo#index'"
end

stage_three do
  # This block is run after the 'stage_two' block.
  # Use this section to finalize the application.
  # Run database migrations or make a final git commit.
end

# A recipe has two parts: the Ruby code and YAML matter that comes
# after a blank line with the __END__ keyword.

__END__

name: mojo
description: "Adding mojo to your application"
author: githubname

category: other
requires: [setup]
run_after: [setup, extras]
```

```
args: -T

config:
  - foo:
      type: multiple_choice
      prompt: What kind of foo do you prefer?
      choices: [ ["Foobar", "foobar"], ["Foobaz", "foobaz"] ]
```

The recipe’s YAML matter must contain keys to set a recipe name (“mojo”), an author (use your GitHub username), and a description (displayed by the `rails_apps_composer list` command).

The “category” directive is used to display an assortment of recipes in groups by the `rails_apps_composer new` command. Use any you like.

The “requires” directive forces the rails_apps_composer program to load any recipes that this recipe depends on.

The “run_after” directive can be given a list of recipes to constrain order of execution. See the section below, Recipe Order, to learn how to set order of execution of recipes.

The “args” directive passes the `-T` or `-O` flags to the `rails new` command to force it to skip Test::Unit or Active Record. This has no affect on application templates and is only useful when an application is generated interactively.

See the section below, Configuring Recipes with YAML, to learn how to set configuration prompts using the YAML format.

Thor and Rails Generator Methods

Thor is a scripting framework for Ruby written by Yehuda Katz. Thor provides utility methods that are useful in manipulating files. Rails generators (used by the `rails new` command) use Thor methods.

You’ll see Thor methods used in the recipes. To learn about the Thor methods used in recipes, refer to the API documentation for Thor::Actions. Here are some Thor methods often used in recipes:

- get
- remove_file
- create_file
- append_to_file
- inject_into_file
- inject_into_class
- gsub_file
- comment_lines
- uncomment_lines

You'll also see Rails Generators methods. To learn about the Rails Generators methods used in recipes, refer to the API documentation for Rails::Generators::Actions. You'll see these methods in the recipes:

- gem
- generate
- git
- route

Rails Wizard Methods

The rails_apps_composer gem inherits several methods from Michael Bleigh's RailsWizard gem:

Input

- ask_wizard(question)
- yes_wizard?(question)
- no_wizard?(question)
- multiple_choice(prompt, [[Label,value], [Label2,value2]])

Output

- say_recipe(name)
- say_custom(tag, text)
- say_wizard(text)

These methods are used to display the prompts that the user sees during the "quiz."

These methods are defined in the file **templates/helpers.erb**.

Rails_apps_composer Methods

The following methods were introduced in rails_apps_composer 2.0 version:

- prefer(key, value)
- copy_from_repo(filename, opts = {})

These methods are defined in the file **templates/helpers.erb**.

The first method is used to test if a preference has been selected. Here's an example:

```
if prefer :authentication, 'devise'
```

The implementation is very simple. It checks if the `prefs{}` hash contains a key/value pair:

```
def prefer(key, value)
  @prefs[key].eql? value
```

```
end
```

The copy_from_repo method is widely used in the recipes. The method can be used with defaults and with options. It takes several forms. Here is the simplest example:

```
copy_from_repo 'public/humans.txt'
```

In its simplest form, it opens a connection to a GitHub repo and copies a file given a path and filename. It uses a default repository.

Another form takes a repository location as an option:

```
copy_from_repo 'app/models/user.rb',
  :repo => 'https://raw.github.com/RailsApps/rails3-devise-rspec-cucumber/master/'
```

This version attempts to copy a file from the given repository and save it with the given filename.

The most powerful form takes a preference value as an option:

```
copy_from_repo 'config/database-mysql.yml', :prefs => 'mysql'
```

In this example, the default repository may contain one or more files with similar names:

- config/database.yml
- config/database-mysql.yml
- config/database-postgresql.yml

If the specified preference is present in the prefs{} hash as a value, the designated file is copied from the default repository. Before the file is saved locally, the preference value is stripped from the filename. In this example, if the user has selected “mysql” as a preference, the method will get a file named **config/database-mysql.yml** from the repository and save it locally as **config/database.yml**. This method makes it possible to store similar files as templates and copy only the one that best matches the selected preferences.

Finally, the copy_from_repo method performs tricks to convert ERB files to Haml or Slim if preferences require it. Here it retrieves an ERB file:

```
copy_from_repo 'app/views/layouts/application.html.erb'
```

No options are required. The copy_from_repo method checks if the user has expressed a preference for Haml or Slim and translates the ERB file accordingly. Only files in the **views/** directory will be converted from ERB to Haml or Slim.

It’s worth examining the source code to understand how the copy_from_repo method works.

```
def copy_from_repo(filename, opts = {})
  repo = 'https://raw.github.com/RailsApps/rails-composer/master/files-v2/'
  repo = opts[:repo] unless opts[:repo].nil?
  if (!opts[:prefs].nil?) && (!prefs.has_value? opts[:prefs])
    return
  end
  source_filename = filename
  destination_filename = filename
  unless opts[:prefs].nil?
    if filename.include? opts[:prefs]
      destination_filename = filename.gsub(/\-#{opts[:prefs]}/, '')
    end
  end
  if (prefer :templates, 'haml') && (filename.include? 'views')
    remove_file destination_filename
    destination_filename = destination_filename.gsub(/.erb/, '.haml')
  end
  if (prefer :templates, 'slim') && (filename.include? 'views')
    remove_file destination_filename
    destination_filename = destination_filename.gsub(/.erb/, '.slim')
  end
  begin
    remove_file destination_filename
    if (prefer :templates, 'haml') && (filename.include? 'views')
      create_file destination_filename, html_to_haml(repo + source_filename)
    elsif (prefer :templates, 'slim') && (filename.include? 'views')
      create_file destination_filename, html_to_slim(repo + source_filename)
    else
      get repo + source_filename, destination_filename
    end
  rescue OpenURI::HTTPError
    say_wizard "Unable to obtain #{source_filename} from the repo #{repo}"
  end
end

def html_to_haml(source)
  html = open(source) {|input| input.binmode.read }
  Haml::HTML.new(html, :erb => true, :xhtml => true).render
end

def html_to_slim(source)
  html = open(source) {|input| input.binmode.read }
  haml = Haml::HTML.new(html, :erb => true, :xhtml => true).render
  Haml2Slim.convert!(haml)
end
```

With an understanding of the methods used in the recipes, you'll be prepared to modify and create recipes.

Recipe Stages

A Rails application is created in stages. The rails_apps_composer program provides methods to manipulate files at appropriate stages of the creation process.

Here are the stages of the creation process:

- the rails_apps_composer program assembles an application template
- `rails new` creates an application

- before_config stage
- the application template prompts the use for preferences (the "quiz")
- the application template modifies a Gemfile and runs bundle install
- stage_two stage
- stage_three stage

The rails_apps_composer recipes are not operative during the second stage of the process. Creating a new application is solely handled by the Rails generators.

The Before_config Stage

Each recipe can contain code that runs before the user is prompted to express preferences. This code will be run before the "quiz."

Recipes supply commands as a block to the before_config method.

Here are examples of procedures to be executed during the before_config stage:

- raise an exception if required conditions are not met
- set preferences that are not optional

In most recipes, nothing is done during the before_config stage.

The Quiz

A simple application template executes a stepwise procedure. Complex application templates, such as those assembled by the rails_apps_composer program, can prompt the user for preferences and execute commands conditionally. Each recipe can request preferences.

The prompt can be driven by inline code. Here's an example:

```
if recipes.include? 'foo'
  prefs[:foo] = multiple_choice "What kind of foo do you prefer?", [ ["Foobar", "foobar"], ["Foobaz", "foobaz"] ]
end
```

The rails_apps_composer program also provides a facility to set up configuration prompts in the form of a YAML file appended to the recipe.

```
config:
  - foo:
      type: multiple_choice
      prompt: What kind of foo do you prefer?
      choices: [ ["Foobar", "foobar"], ["Foobaz", "foobaz"] ]
```

See the section below, Configuring Recipes with YAML, to learn how to set configuration prompts using the YAML format.

Adding a Gem

After obtaining the user's preferences, the application template will add gems to a Gemfile.

There are several ways to tell rails_apps_composer to add a gem:

- respond to a prompt and enter the name of a gem
- include the name of a gem as a parameter in the defaults file
- modify the *gems* recipe to add a gem
- create a custom recipe that adds a gem

Here's a simple example showing how a recipe can add a gem to the Gemfile:

```
gem 'foobar'
```

Here's a complex example from a recipe:

```
gem 'factory_girl_rails', '>= 3.5.0', :group => [:development, :test] if prefer :fixtures, 'factory_girl'
```

In this example, the factory_girl_rails gem is added to the Gemfile if the user has specified 'factory_girl' as a preference for 'fixtures'. Extra arugments specify that the gem will only be used in development and test environments. You are not required to specify a gem version. However, for the RailsApps example applications, we use optimistic version constraint (using the ">=" operator) to prevent use of older versions of gems that may already be installed but not compatible. We allow any newer version of the gem.

The After_bundler Stage

After the Gemfile is prepared, the application template runs `bundle install`.

The rails_apps_composer program uses the `stage_two` method to postpone execution of commands until after `bundle install` runs.

Most recipes will supply commands (as a block) to the `stage_two` method. The application template executes all code in `stage_two` blocks immediately after running `bundle install`.

Here are examples of procedures to be executed during the `stage_two` stage:

- `generate` commands to run generators after a gem is installed
- code to create models, views, controllers, or views

Most of your code will execute in the `stage_two` stage.

The After_everything Stage

Sometimes you will have code that must execute after everything else is ready.

The rails_apps_composer program provides an stage_three method that postpones execution of commands until after stage_two runs.

Like stage_two, recipes supply commands as a block to the stage_three method.

Here are examples of procedures to be executed during the stage_three stage:

- database migrations
- seeding a database
- removing unnecessary files and whitespace
- final commit to a git repository

The stage_three stage is the place to clean up and finalize your application.

Recipe Order

Recipe order is critical to successful execution of an application template. You must give careful thought to order of execution when assembling recipes.

For example, if a recipe attempts to manipulate a User model but the User model has not yet been created, the application template will fail.

Just to review: Within a recipe, you specify order of execution by placing code in blocks supplied to the before_config, stage_two, and stage_three methods. The methods before_config, stage_two, and stage_three execute in sequence. Code that is not placed in one of these blocks executes between the before_config and stage_two method calls.

But among recipes, what determines the order in which recipes execute?

The sequence in which you list the recipes determines the order in which the recipes execute. If you specify recipes at the interactive prompt, the order in which you enter recipes determines precedence. If you provide a list of recipes (using the -r flag) when you build an application or an application template, the precedence is determined by the list order.

In general, the easiest way to assure that recipes are executed in the order you desire is to order the list of recipes and use the the -r flag when you run the rails_apps_composer program. For example:

```
$ rails_apps_composer new myapp -r setup readme gems extras
```

The example will execute all recipes in sequence to prompt the use for preferences, then execute each stage_two block, then run each stage_three block in the sequence you've listed the recipes.

You can also constrain execution order by setting the run_after directive in the YAML matter appended to a recipe. See the section below, Configuring Recipes with YAML, to learn how to constrain execution order using the YAML format.

You can force recipes to be loaded by setting the requires directive in the YAML matter.

The YAML Matter

At a minimum, every recipe must contain YAML matter that specifies the recipe name, description and author:

```
__END__

name: mojo
description: "Adding mojo to your application"
author: githubname
```

The __END__ keyword marks the transition from Ruby code to data. Everything that follows is input as data and parsed by a YAML interpreter.

Descriptive data is entered as key/value pairs including name, description, and author.

Configuring Recipes with YAML

Any recipe can ask the user for preferences and execute commands conditionally.

The rails_apps_composer program provides a facility to set up configuration prompts in the form of a YAML file appended to the recipe.

A recipe that sets up configuration prompts looks like this:

```
__END__

name: mojo
description: "Adding mojo to your application"
author: githubname

category: other
requires: [setup]
run_after: [setup, extras]

config:
  - foo:
      type: multiple_choice
      prompt: What kind of foo do you prefer?
      choices: [ ["Foobar", "foobar"], ["Foobaz", "foobaz"] ]
```

The requires directive forces the rails_apps_composer program to load any other recipes that are required by the recipe.

Order of execution can be constrained with the run_after directive by passing an array of recipe names that must be executed before this recipe will be allowed to run.

Finally, the config schema accepts a list of hashes that prompt the user for preferences.

Complex YAML Example

The YAML matter can be a powerful tool for configuring recipes.

Here is an example of a complex configuration schema:

```
__END__

name: flight_test
description: "Test your application in space"
author: githubname

category: testing
requires: [testing]
run_after: [testing]

config:
  - space_test:
      type: boolean
      prompt: Do you want to test your application in space?
  - mars_test:
      type: boolean
      prompt: Do you also want to test your application on Mars?
      if: space_test
      if_recipe: mars_lander
  - test_count:
      type: string
      prompt: How many times would you like to repeat the test?
      if: space_test
  - orbit:
      type: multiple_choice
      prompt: "What orbit do you want?"
      choices: [ [Low Earth orbit, leo], [Sun-synchronous, spy], [Geostationary, gps] ]
      if: space_test
```

In this example, a *testing* recipe is required and the recipe is forced to run after the *testing* recipe.

The keys in the list of hashes will become keys in the `config{}` hash available to the recipe.

For example, the boolean `config['space_test']` will be true or false:

```
if config['space_test']
  say_wizard "running a space test"
end
```

The `if: space_test` condition will insure that other prompts only appear if the user has answered “yes” to the first prompt.

The `if_recipe: mars_lander` condition will only prompt the user for a reply if a “mars_lander” recipe has been included in the list of recipes.

The “test_count” prompt takes a string as a response. The string is available as:

```
if config['space_test']
  say_wizard "will run the test #{config['test_count']} times"
end
```

The “orbit” prompt will appear as a multiple choice question:

```
question  What orbit do you want?
        1)  Low Earth orbit
        2)  Sun-synchronous
        3)  Geostationary
  flight_test  Enter your selection:
```

The response is shown in this example:

```
if config['space_test']
  say_wizard "will run the test in #{config['orbit']} orbit"
end
```

You might use a case statement to read recipe configuration parameters and set global preferences:

```
case config['orbit']
  when 'leo'
    prefs[:test_orbit] = 'leo'
  when 'spy'
    prefs[:test_orbit] = 'spy'
  when 'gps'
    prefs[:test_orbit] = 'gps'
end
```

Keep in mind that the `config` hash is only locally available to the recipe. The `prefs{}` hash is available to all the recipes.

Architecture

This section describes the architecture of the rails_apps_composer program. Learn about the architecture to understand how the rails_apps_composer program works.

Application templates do not create a new Rails application. That job is the prerogative of the `rails new` command. Application templates modify a simple Rails application that is created by the `rails new` command.

The rails_apps_composer gem exploits the application template functionality introduced in Rails 2.3. The RailsCast App Templates in Rails 2.3 from February 2009 offers an introduction to Rails application templates.

The rails_apps_composer program assembles an application template programmatically from “recipes” which are files of Ruby code. When you ask the rails_apps_composer gem to generate an application, the program prompts for recipe names and preferences, assembles a temporary application template, and then runs the `rails new` command supplying the application template file. When you ask the rails_apps_composer gem to generate and save a template, it assembles the template, saves the file, and halts before generating an application.

Recipe Local Variables

Local recipe configuration parameters and global preferences are used to set state for the application template.

Each recipe has a `config{}` hash that stores preferences entered from prompts parsed from the recipe's YAML matter. If the recipe has a YAML section that includes:

```
config:
  - foo:
      type: multiple_choice
      prompt: What kind of foo do you prefer?
      choices: [ ["Foobar", "foobar"], ["Foobaz", "foobaz"] ]
  - qux:
      type: boolean
      prompt: Include a Qux Flux Modulator?
```

The configuration parameter can be read and used to set a global `prefs{}` key/value pair:

```
case config['foo']
  when 'foobar'
    prefs[:foo] = 'foobar'
  when 'foobaz'
    prefs[:foo] = 'foobaz'
end
```

Or it can be used for conditional execution of a recipe procedure:

```
if config['qux']
  copy_from_repo 'lib/qux.rb'
end
```

Global Variables

Three data structures are available to all the recipes:

- `recipes[]` array
- `gems[]` array
- `prefs{}` hash

Recipes

The `recipes[]` array is a list of recipes to include in the application template. It can be set from a defaults file, from the command line (with the `-r` flag), or from an interactive prompt.

Test for the presence of a recipe with:

```
if recipes.include? 'email'
  ...
end
```

A convenience method is also available, which does the same thing as the code above:

```
recipe?(name)
  ...
end
```

Every recipe is optional, so it is wise to test for the presence of a recipe before executing any code that requires the recipe.

Prefs

The `prefs{}` hash makes the user's preferences available to all recipes. It can be set from a defaults file or programmatically by any recipe. You'll often set a key/value pair in the `prefs{}` hash after reading a local `config` value solicited by a recipe.

The `prefer(key, value)` method lets you test for a preference:

```
if prefer :authentication, 'devise'
  ...
end
```

Note that the hash keys are symbols (preceded by a colon).

Gems

The `gems[]` array is a list of gems to add to the Gemfile. It can be set from a defaults file or from an interactive prompt. The *gems* recipe adds each gem named in the array to the Gemfile. The array is available to other recipes but seldom used outside of the *gems* recipe.

Templates

Three template files are the basis for generating an application template:

- **templates/layout.erb**
- **templates/recipe.erb**
- **templates/helpers.erb**

Layout for an Application Template

The file **templates/layout.erb** is the "template for an application template." If you compare the **templates/layout.erb** file with an application template generated by the rails_apps_composer program, you'll see that the **templates/layout.erb** file contains placeholders and variables that are set by the program at run time.

The file **templates/layout.erb** provides the structure for the application template. It is organized in sections:

- Initial Setup
- Autoload Modules/Classes
- Recipes
- Run ‘Bundle Install’
- Run ‘After Bundler’ Callbacks
- Run ‘After Everything’ Callbacks

If you wanted to put your own name on every application template generated by the rails_apps_composer program, you’d modify the **templates/layout.erb** file.

Recipe Partial for an Application Template

The file **templates/recipe.erb** is included in the layout template for each recipe requested by the user.

Variables in the ‘recipe’ partial are set by a RailsWizard::Recipe instance.

A `configs[]` array provides access to the configuration parameters set by the user’s responses. These configuration parameters are only accessible within the context of the current recipe. Use them to set global `prefs{}` key/value pairs if the configuration parameters need to be available to other recipes.

Helpers Partial for an Application Template

The file **templates/helpers.erb** is included in the layout template.

This file is essential for defining methods that will be used by any recipes.

These methods are used to display the prompts that the user sees during the “quiz”:

- `ask_wizard(question)`
- `yes_wizard?(question)`
- `no_wizard?(question)`
- `multiple_choice(prompt, [ [Label,value], [Label2,value2] ])`
- `say_recipe(name)`
- `say_custom(tag, text)`
- `say_wizard(text)`

This method is used to determine if the user has expressed a preference:

- `prefer(key, value)`

This method copies a file from a remote repository:

- `copy_from_repo(filename, opts = {})`

These methods are described in more detail above in the sections Rails Wizard Methods and Rails_apps_composer Methods.

Gem Internals

The **lib/** directory contains the code that drives the rails_apps_composer program.

Much of the code that drives the program is duplicated from Michael Bleigh's RailsWizard.

RailsWizard::Command interprets commands entered on the command line.

RailsWizard::Config creates the prompts defined by a recipe's YAML matter.

RailsWizard::Recipe is an object populated by a recipe's YAML matter that is used to define variables in the **templates/recipe.erb** file.

RailsWizard::Recipes is an object that contains all the recipes.

RailsWizard::Template defines variables in the **templates/layout.erb** file.

RailsWizard::Diagnostics was introduced in rails_apps_composer 2.0 to provide an internal database of combinations of recipes and preferences that work together.

If you are interested only in creating or modifying recipes, you won't need to examine the gem internals.

Defaults File

If you frequently use the same recipes and preferences, create a defaults file and produce the same application every time.

Generate an application from defaults:

```
$ rails_apps_composer new myapp --defaults=my_defaults.yaml
```

Or generate an application template from defaults:

```
$ rails_apps_composer template ~/Desktop/template.rb --defaults=my_defaults.yaml
```

Here's the format of a typical defaults file:

```
recipes:
- setup
- readme
- gems
- testing
- email
- models
- controllers
- views
- routes
- frontend
- database
- extras

prefs:
  :dev_webserver: webrick
  :prod_webserver: same
  :database: sqlite
  :templates: erb
  :tests: rspec
  :integration: cucumber
  :fixtures: factory_girl
  :frontend: none
  :form_builder: none
  :email: gmail
  :authentication: devise
  :devise_modules: default
  :authorization: none

gems:
- heroku
- nokogiri
- kaminari

args:
  :skip_test_unit: yes
  :skip_active_record: false
```

Recipes

The "recipes" schema is used to initialize the `recipes[]` array. When you run the command with the `--defaults` option and no recipes specified on the command line, you'll see the prompt, "Which recipe would you like to add? (blank to finish)". If you want to use only your default recipes, just hit return. If you want to add a recipe for the app you are generating, just type the recipe name at the prompt.

Prefs

The "prefs" schema is used to initialize the `prefs{}` hash. Add key/value pairs for each preference you wish to set. Note the key is a Ruby symbol, not a string (it begins with a colon). You'll have to examine the recipe files to determine what keys (such as `dev_webserver`) are used to specify preferences. If there is no preference specified in the defaults file, the rails_apps_composer program will ask you to respond with a preference. That way, you can give yourself flexibility if you don't always have the same preference.

Gems

The "gems" schema is used to initialize the `gems[]` array. This is a list of strings. You must select the *gems* recipe for this list to function. The *gems* recipe iterates through this list and adds each gem to the Gemfile. The rails_apps_composer program doesn't check if the gem exists; it assumes you've entered the name of an available gem. Whether or not you've listed gems in the defaults file, the rails_apps_composer program will prompt you to enter any additional gems you would like.

Args

The "args" schema sets `-T` or `-O` arguments to skip Test::Unit or Active Record when you generate an application interactively. Setting `-T` or `-O` arguments has no affect on application templates.

Hidden Preferences

Any recipe can respond to preferences that are not set from an interactive prompt if the preference setting is included in the defaults file. In general, if it is likely that any developer would like to make a preference choice, you should provide a prompt. However, there may be some preference settings that are not needed by most developers. These "obscure" preferences can be set in the defaults file. You'll know about the preferences, as will any developer who examines the recipe closely. But you'll eliminate the need to query every developer for a preference that may be seldomly used.

At this time, there is one hidden preference that changes the behavior of the rails_apps_composer program. Some developers like to install gems to an alternate location (by default, bundler installs your gems to a system location such as **.bundle/gems**). If your defaults file contains the preference `:bundle_path: some_directory`, the rails_apps_composer program will supply the alternative bundle_path when running `bundle install`.

Additional Documentation

This is the primary documentation for the rails_apps_composer gem.

About Rails Application Templates

Cooking Up A Custom Rails 3 Template (11 Oct 2010) by Andrea Singh
Rails Application Templates (16 Sept 2010) by Collin Schaafsma
Application templates in Rails 3 (18 Sept 2009) by Ben Scofield
Railscasts: App Templates in Rails 2.3 (9 Feb 2009) by Ryan Bates
Rails templates (4 Dec 2008) by Pratik Naik

Issues

Any issues? Please create an issue on GitHub. Reporting issues (and patching!) helps everyone.

When you generate an application, the application's **README.textile** file will contain a diagnostics report. Please include the diagnostics report when you create an issue on GitHub.

Credits

Daniel Kehoe maintains this gem as part of the RailsApps Project.

This gem is derived from Michael Bleigh's RailsWizard gem. The original idea and the innovative implementation is the work of Michael Bleigh.

Please see the CHANGELOG for a list of contributors.

Is the gem useful to you? Follow the project on Twitter: @rails_apps. I'd love to know you were helped out by the gem.

MIT License (with restriction)

+ Add a custom footer

Pages 2

Find a Page…

- **Home**
- **tutorial rails apps composer**

+ Add a custom sidebar

Clone this wiki locally

https://github.com/Rails

Clone in Desktop