Un enfoque práctico del desarrollo guiado por pruebas

Pruebas con RSpec

everyday Rails

Aaron Sumner

Everyday Rails Pruebas con RSpec (Edición en Español)

Un enfoque práctico del desarrollo guiado por pruebas

Aaron Sumner

Este libro está a la venta en https://leanpub.com/everydayrailsrspec-es

Esta versión se publicó en 2026-04-18

Éste es un libro de Leanpub. Leanpub anima a los autores y publicadoras con el proceso de publicación. Lean Publishing es el acto de publicar un libro en progreso usando herramientas sencillas y muchas iteraciones para obtener retroalimentación del lector hasta conseguir el libro adecuado.

¡Tuitea sobre el libro!

Por favor ayuda a Aaron Sumner hablando sobre el libro en Twitter!

El tuit sugerido para este libro es:

I'm learning to test my Rails apps with Everyday Rails Testing with RSpec.

El hashtag sugerido para este libro es #everydayrailsrspec.

Descubre lo que otra gente dice sobre el libro haciendo clic en este enlace para buscar el hashtag en Twitter:

#everydayrailsrspec

Índice general

Prefacio a esta edición

Gracias por echarle un vistazo a *Everyday Rails Testing with RSpec*, ahora actualizado para Ruby on Rails 8.1. Seré honesto: a medida que mi trabajo diario ha evolucionado, ya no escribo tanto Ruby como antes. ¡Pero mi mente sigue pensando en Ruby, y Rails sigue siendo mi primera opción para muchos proyectos!

Desde la edición anterior, sin embargo, tomé la decisión de cambiar el enfoque de Everyday Rails, el blog del que nació este libro en 2012, para cubrir el desarrollo de software de manera más amplia. El sitio, renombrado y con nueva imagen, se llama Left of the Dev.

Aunque hoy en día no escribo Ruby a diario, sigue resultándome fascinante ver cómo el ecosistema de Ruby —incluyendo Rails y RSpec— influye en lo que yo creo que debería *simplemente funcionar* en el desarrollo de software. ¡Los que trabajamos con Ruby estamos mimados por la maravillosa atención que se le presta a la experiencia del desarrollador!

Sigue siendo un gran momento para ser desarrollador de Rails. Y si tu trabajo es en otro lenguaje de programación, no hay problema. RSpec sigue siendo, en mi opinión, el mejor framework de pruebas en su categoría, y una excelente manera de aprender las pruebas como práctica.

Espero que disfrutes esta edición del libro. Cuídense mucho por ahí, y sigan escribiendo software por diversión, por el bien y, a veces, por dinero.

Aaron Sumner
Saint Louis, MO, USA
Abril, 2026

1. Introducción

Ruby on Rails y las pruebas automatizadas van de la mano. Rails incluye un framework de pruebas integrado. Cuando usas los generadores, crea automáticamente archivos de prueba base listos para que los rellenes con tus propias pruebas. Eso es bastante genial, si me preguntas.

Sin embargo, muchas personas que desarrollan en Rails todavía no prueban sus proyectos en absoluto, o en el mejor de los casos, solo añaden unas pocas specs simbólicas sobre cosas básicas que puede que ni siquiera sean útiles ni informativas. Quizás trabajar con Ruby o con frameworks web dogmáticos ya es un concepto suficientemente novedoso, y añadir una capa extra de trabajo parece exactamente eso: *¡trabajo extra!* O tal vez hay una sensación de falta de tiempo — dedicar tiempo a escribir pruebas se lo quita a escribir las funcionalidades que nuestros clientes o jefes exigen. O quizás el hábito de definir *probar* como la práctica de hacer clic en enlaces en el navegador es simplemente demasiado difícil de romper.

Yo he estado ahí. Como ingeniero, tengo problemas que resolver. Y, por lo general, encuentro soluciones a esos problemas construyendo software. Llevo desarrollando aplicaciones web desde 1995. Durante mucho tiempo, trabajé como desarrollador en solitario en proyectos del sector público con presupuesto mínimo. Salvo una exposición estructurada a BASIC de niño, algo de C++ en la universidad y una semana perdida de formación en Java en mi segundo trabajo de adulto fuera de la universidad, nunca tuve una educación formal de verdad en desarrollo de software. De hecho, no fue hasta 2005, cuando ya había tenido suficiente de hackear código PHP feo al estilo espagueti, que busqué una mejor manera de escribir aplicaciones web.

Había mirado Ruby antes, pero nunca le había encontrado un uso serio hasta que Rails empezó a cobrar fuerza. Había mucho que aprender — un nuevo lenguaje, una *arquitectura* real y un enfoque más orientado a objetos. Aun con todos esos nuevos desafíos, era capaz de crear aplicaciones complejas en una fracción del tiempo que me llevaba con mis esfuerzos anteriores sin framework. ¡Estaba enganchado!

Pruebas con confianza

Pero los primeros libros y tutoriales de Rails se centraban más en la velocidad de desarrollo (¡construye un blog en 15 minutos!) que en buenas prácticas como las pruebas. Si las trataban, por

lo general las reservaban para un capítulo hacia el final. Los recursos educativos más recientes sobre Rails han abordado esta carencia y demuestran cómo probar aplicaciones desde el principio. Hoy existen innumerables libros *específicamente* sobre el tema de las pruebas. Pero sin un enfoque sólido hacia el lado de las pruebas, muchos desarrolladores — especialmente aquellos en una situación similar a la que yo estaba — pueden encontrarse sin una estrategia de pruebas coherente. Si es que hay pruebas, puede que no sean fiables ni significativas. Ese tipo de pruebas no generan *confianza del desarrollador.*

Mi primer objetivo con este libro es presentarte una estrategia coherente que funciona para *mí* — una que tú puedas, con suerte, adaptar para que también funcione de forma coherente para *ti.* Si lo logro, al leer este libro *probarás con confianza.* Podrás hacer cambios en tu código sabiendo que tu suite de pruebas te cubre las espaldas y te avisará si algo se rompe.

¿Por qué RSpec?

No tengo nada en contra de otros frameworks de pruebas de Ruby. Si el framework MiniTest predeterminado te ayuda a escribir suites de pruebas sostenibles con confianza, ¡genial!

Para mí, la capacidad de RSpec para producir specs que son legibles sin resultar excesivamente engorrosas es lo que me convence. Hablaré más sobre esto más adelante en el libro, pero he descubierto que con un poco de orientación, incluso la mayoría de las personas no técnicas pueden leer una spec escrita en RSpec y entender qué está pasando. Es tan expresivo que usar RSpec para describir cómo espero que se comporte mi software se ha convertido en algo completamente natural. La sintaxis fluye de mis dedos y es agradable de leer en el futuro cuando estoy haciendo cambios en mi software.

Mi segundo objetivo con este libro es ayudarte a sentirte cómodo con la funcionalidad y la sintaxis de RSpec que es más probable que uses de forma habitual. RSpec es un framework complejo, pero como muchos sistemas complejos, a menudo te encontrarás usando el 20 por ciento de la funcionalidad disponible para el 80 por ciento de tu trabajo. Con eso en mente, esta no es una guía completa de RSpec ni de librerías complementarias como Capybara. En cambio, se centra en las herramientas que he usado durante años para probar mis propias aplicaciones Rails. También te presentará patrones comunes para que, cuando te encuentres con un problema que no está cubierto en el libro, puedas buscar soluciones de forma inteligente y no te quedes atascado.

Quién debería leer este libro

Si Rails es tu primera incursión en un framework de aplicaciones full-stack y tu experiencia de programación anterior no incluía pruebas de ningún tipo, este libro te ayudará, con suerte, a empezar. Si eres *muy* nuevo en Rails, puede que te resulte beneficioso revisar material sobre desarrollo y pruebas básicas. Mi favorito es *Agile Web Development with Rails 8* de Sam Ruby; a mucha gente también le gusta el *Rails Tutorial* de Michael Hartl. Mi libro asume que ya tienes algunas habilidades básicas de Rails a tus espaldas. En otras palabras, no te enseñará a usar Rails ni te proporcionará una introducción desde cero a las herramientas de prueba integradas en el framework. En cambio, vamos a instalar RSpec y algunos extras para que el proceso de pruebas sea lo más fácil posible de entender y gestionar. Así que si eres nuevo en Rails, echa un vistazo a uno de esos recursos primero y luego vuelve a este libro.

Si llevas un tiempo desarrollando en Rails pero las pruebas siguen siendo un concepto ajeno para ti, ¡este libro es para ti! Yo estuve en tu misma posición durante mucho tiempo, y las técnicas que compartiré aquí me ayudaron a mejorar mi cobertura de pruebas y a pensar más como un desarrollador guiado por pruebas. Espero que hagan lo mismo por ti.

En concreto, deberías tener un buen manejo de:

- Las convenciones de las aplicaciones Model-View-Controller del lado del servidor, tal como se usan en Rails
- Bundler para la gestión de dependencias de gems
- Cómo trabajar con la línea de comandos de Rails

Si ya estás familiarizado con MiniTest, o incluso con RSpec, y ya tienes un flujo de trabajo consolidado que te da confianza, puede que puedas ajustar algunos aspectos de tu enfoque para probar tus aplicaciones. Espero que también aprendas de mi enfoque con criterio propio sobre las pruebas, y de cómo pasar de simplemente probar tu código a hacerlo con intención.

Este *no* es un libro sobre teoría general de pruebas, y tampoco profundiza demasiado en los problemas de mantenimiento y rendimiento que pueden ir acumulándose en el software heredado con el tiempo. Otros libros pueden ser más útiles en esos temas.

Mi filosofía sobre las pruebas

¿Qué tipo de pruebas es mejor: las unitarias o las de integración? ¿Debería practicar el desarrollo guiado por pruebas o el desarrollo guiado por comportamiento (y cuál es la diferencia, de todas

formas)? ¿Debería escribir mis pruebas antes de escribir el código, o después? ¿O debería siquiera molestarme en escribir pruebas?

Debatir sobre la forma *correcta* de probar tu aplicación Rails puede desatar encendidas disputas entre programadores. Sí, existe una forma correcta de hacer pruebas, pero en mi opinión, hay distintos grados de *correcto* cuando se trata de este tema. Mi enfoque se basa en las siguientes creencias fundamentales:

- Las pruebas deben ser confiables.
- Las pruebas deben ser fáciles de escribir.
- Las pruebas deben ser fáciles de entender: hoy y en el futuro; por ti y por tus colaboradores.

En resumen: las pruebas deben darte *confianza* como desarrollador de software. Si tienes en cuenta estos tres factores en tu enfoque, estarás muy bien encaminado hacia una suite de pruebas sólida para tu aplicación, sin mencionar que te convertirás en un auténtico practicante del Desarrollo Guiado por Pruebas.

Sí, hay algunas concesiones; en particular:

- No nos estamos enfocando en la velocidad, aunque hablaremos de ello más adelante.
- No nos estamos enfocando en un código excesivamente DRY en nuestras pruebas. Pero en las pruebas, eso no es necesariamente algo malo. También hablaremos de esto.

Al final, sin embargo, lo más importante es que *tendrás buenas pruebas*, y las pruebas confiables y comprensibles son una excelente manera de empezar, aunque no estén tan optimizadas como podrían estarlo. Este es el enfoque que finalmente me ayudó a superar el obstáculo entre escribir mucho código de aplicación, llamar "pruebas" a una ronda de clics en el navegador, subir a producción y esperar lo mejor. Hay una forma mejor: aprovechar una suite de pruebas completamente automatizada y usar las pruebas para guiar el desarrollo y rastrear posibles errores y casos límite *antes* de que los usuarios los encuentren.

Y ese es el enfoque que adoptaremos en este libro.

Cómo está organizado el libro

En *Testing Rails with RSpec* te guiaré a través del proceso de llevar una aplicación básica de Rails de no tener ninguna prueba a tener una cobertura respetable con RSpec. El libro cubre Rails 8.1 y

RSpec 3.13 (a través de rspec-rails 8.0). Muchos de los conceptos se aplican a versiones anteriores y posteriores de cada uno, aunque con una sintaxis ligeramente diferente.

El libro comienza con la instalación de RSpec en una aplicación Rails existente. A partir de ahí, construimos una suite de pruebas poco a poco, comenzando con pruebas pequeñas y aisladas y avanzando hacia escenarios de prueba más amplios y complejos. Este es el mismo proceso paso a paso que usé para mejorar en las pruebas de mi propio software. Te recomiendo encarecidamente trabajar en los ejercicios dentro de tus propias aplicaciones: una cosa es seguir un tutorial, y otra muy distinta es aplicar lo que aprendes a tu propia situación. En este libro no construiremos una aplicación juntos, sino que exploraremos patrones y técnicas de código. ¡Toma esas técnicas y mejora tus propios proyectos!

Descarga del código de ejemplo

Puedes obtener el código de ejemplo en https://leftofthe.dev/rspecbook. El archivo Zip proporcionado contiene instantáneas de la aplicación de ejemplo y su suite de pruebas a medida que crece, capítulo a capítulo. Por ejemplo, el directorio *02-models* incluye la propia aplicación Rails y las pruebas añadidas en el capítulo 2.

Convenciones de código

Estoy usando la siguiente configuración para esta aplicación:

- **Rails 8.1:** La versión más reciente de Rails es el principal foco de este libro. Que yo sepa, la mayoría de las técnicas que utilizo se aplicarán a cualquier versión de Rails desde la 7.2 en adelante. Es posible que algunos ejemplos de código no funcionen exactamente igual en tu caso, pero haré todo lo posible por indicarte dónde podrían existir diferencias.
- **Ruby 4.0:** Cualquier versión de Ruby 4 debería funcionar. Ruby 3.2 o superior es el mínimo requerido por rspec-rails 8.0.
- **RSpec 3.13 (rspec-rails 8.0):** RSpec 3.0 fue lanzado en la primavera de 2014 y ha permanecido bastante estable desde entonces. La gem rspec-rails ahora sigue el versionado de Rails (de ahí el salto de la versión 6.x a la 8.x). Las versiones anteriores usaban una sintaxis significativamente diferente que no cubriremos en esta edición del libro.

Si algo es específico de estas versiones, haré todo lo posible por señalarlo. Si estás trabajando con una versión más antigua de Rails, RSpec o Ruby, las versiones anteriores del libro están

disponibles como descargas gratuitas a través de Leanpub al adquirir esta edición. No son idénticas en cuanto a funcionalidades, pero con suerte deberías poder ver algunas de las diferencias básicas.

De nuevo, **¡este libro no es un tutorial tradicional!** El código que se incluye aquí no pretende guiarte en la construcción de una aplicación. Está aquí para ayudarte a entender y aprender patrones y hábitos de pruebas para aplicar a tus propias aplicaciones Rails. En otras palabras, puedes copiar y pegar, pero probablemente no te sirva de mucho a largo plazo.

Discusión y erratas

He dedicado mucho tiempo y esfuerzo a asegurarme de que *Testing Rails with RSpec* sea lo más libre de errores posible, pero puede que encuentres algo que se me haya pasado por alto. Si ese es el caso, dirígete a la sección de issues del código fuente en GitHub para compartir un error o pedir más detalles: https://github.com/everydayrails/rspec-sample-rails-8.1/issues

Una nota sobre las versiones de gemas

Las versiones de gemas utilizadas en este libro y en la aplicación de ejemplo son las actuales en el momento en que escribo esta edición de Rails 8.1, en 2026. Por supuesto, cualquiera de ellas puede actualizarse con frecuencia, así que estate atento a ellas en Rubygems.org, GitHub y tus fuentes de noticias favoritas sobre Ruby.

Una nota sobre el estilo

Muchos de los ejemplos de código incluidos en este libro están basados en código creado por generadores. Como resultado, es posible que veas ejemplos con estilos inconsistentes — por ejemplo, el uso de comillas simples y dobles en el mismo archivo. Con el fin de preservar las diferencias entre lo que se genera y lo que importa para aprender RSpec, he optado por dejar los estilos del código generado tal como están, pero en general seguiré las convenciones definidas por Standard Ruby.

Sobre la aplicación de ejemplo

¡Conoce TasteDrivenDishes, el último sitio social para encontrar y compartir tus recetas favoritas con usuarios de todo el mundo! Será mejor que tengamos una suite de pruebas en marcha antes de que el sitio llegue a la portada de Hacker News — es mejor que seamos nosotros quienes los detectemos (y corrijamos) antes que nuestros usuarios e inversores.

Para empezar, TasteDrivenDishes incluye las siguientes funcionalidades:

- Un usuario puede explorar recetas y filtrar por categoría.
- Un usuario puede crear una cuenta para añadir sus propias recetas.
- Un usuario con sesión iniciada puede marcar recetas como favoritas.
- La cuenta de un usuario tiene un avatar, proporcionado por el servicio Dicebear.
- Un desarrollador puede acceder a una API pública para desarrollar aplicaciones cliente externas.

Hasta este punto, he evitado intencionadamente escribir pruebas para la aplicación (consulta la carpeta *01-untested* en la descarga del código de ejemplo). Esto significa que tengo un directorio *test* lleno de archivos de prueba intactos y preparación de datos. Podría ejecutar `bin/rails test`, y quizás algunas de estas pruebas incluso pasarían. Pero como este es un libro sobre RSpec, eliminaremos esta carpeta, configuraremos Rails para usar RSpec en su lugar, y construiremos una suite de pruebas fiable. Eso es lo que abordaremos en este libro.

Lo primero es lo primero: necesitamos configurar la aplicación para que reconozca y use RSpec. ¡Empecemos!

2. Configuración de RSpec

TasteDrivenDishes está *funcionando* en este momento. Al menos *creemos* que está funcionando. Nuestra única prueba de ello es que navegamos por los enlaces, creamos algunas cuentas y recetas de prueba, y agregamos y editamos datos a través del navegador web.

¿Lo lanzamos así nomás, verdad?

Por supuesto, este enfoque no escala a medida que agregamos funcionalidades. Antes de continuar, hagamos una pausa en el desarrollo de funcionalidades y agreguemos una *suite de pruebas automatizadas*, con RSpec en su núcleo. A lo largo del resto de este libro, iremos agregando cobertura a TasteDrivenDishes, comenzando con RSpec y añadiendo otras bibliotecas de prueba según sea necesario para completar la suite.

Hubo un tiempo en que se necesitaba un esfuerzo considerable para lograr que RSpec y Rails funcionaran juntos. Eso ya no es el caso, pero aún necesitaremos instalar algunas cosas y ajustar algunas configuraciones antes de empezar a agregar specs.

En este capítulo, completaremos las siguientes tareas:

- Comenzaremos usando Bundler para instalar RSpec.
- A continuación, verificaremos si existe una base de datos de prueba y la instalaremos si es necesario.
- Por último, ¡configuraremos RSpec para ejecutar la suite de pruebas!

Para seguir el código de este capítulo, comienza con la carpeta *01-untested* del código de ejemplo. Consulta la carpeta *02-setup* para ver el código completado de este capítulo. Consulta *"Downloading the sample code"* en el Capítulo 1 para obtener instrucciones de descarga.

Dependencias

Como RSpec no está incluido en una aplicación Rails predeterminada, necesitaremos tomarnos un momento para instalarlo. Usaremos Bundler para agregar la dependencia. Si aún no tienes una terminal de línea de comandos abierta en la aplicación, abre una ahora. Luego, en el indicador de la línea de comandos, escribe:

```
bundle add rspec-rails --version "~> 8.0.4" --group "development, test"
```

Ten en cuenta que solo estamos instalando RSpec para usarlo en los entornos de *desarrollo* y *prueba* de la aplicación. No se instalará al desplegar la aplicación en producción. También hemos fijado la versión para que Bundler instale cualquier versión de la gem `rspec-rails` igual o mayor a 8.0.4, pero no la 8.1 ni versiones más recientes.

Técnicamente, estamos instalando la biblioteca `rspec-rails`, que incluye `rspec-core` y algunas otras gems independientes. Si estuvieras usando RSpec para probar una aplicación Ruby que no usa Rails, podrías instalar estas gems de forma individual. `rspec-rails` las agrupa en una sola instalación conveniente, junto con algunas facilidades específicas de Rails de las que comenzaremos a hablar pronto.

Nuestra aplicación ahora cuenta con el primer bloque fundamental necesario para establecer una suite de pruebas sólida. Lo siguiente: crear nuestra base de datos de prueba.

Base de datos de prueba

Con el propósito de enseñarte sobre RSpec sin demasiada carga adicional, TasteDrivenDishes usa SQLite como backend de base de datos.

Si estás agregando specs a una aplicación Rails existente, es posible que ya tengas una base de datos de prueba en tu computadora. Si no es así, aquí te explicamos cómo agregar una.

Abre el archivo *config/database.yml* para ver con qué bases de datos está lista para comunicarse tu aplicación. Si no has realizado ningún cambio en el archivo, deberías ver algo como lo siguiente:

config/database.yml

```
test:
  <<: *default
  database: storage/test.sqlite3
```

Rails 8.1 almacena las bases de datos SQLite en el directorio *storage/* en lugar del directorio *db/* utilizado en versiones anteriores. Si estás usando MySQL o PostgreSQL, la configuración de la base de datos se verá diferente — consulta la guía de Rails *Configuring Rails Applications* para más detalles.

Para asegurarte de que haya una base de datos con la que comunicarse, ejecuta la siguiente tarea rake:

```
$ bin/rails db:create:all
```

Si aún no tenías una base de datos de prueba, ahora ya la tienes. Si ya tenías una, la tarea `rails` te informa amablemente que la base de datos ya existe — no es necesario preocuparse por eliminar accidentalmente una base de datos anterior. Ahora configuremos el propio RSpec.

Configuración de RSpec

Ahora podemos agregar una carpeta spec a nuestra aplicación y añadir una configuración básica de RSpec. Instalaremos RSpec con la siguiente directiva de línea de comandos:

```
$ bin/rails generate rspec:install
```

Y el generador muestra:

```
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

Ahora tenemos un archivo de configuración para RSpec (*.rspec*), un directorio para nuestros archivos de spec a medida que los vayamos creando (*spec*), y dos archivos auxiliares donde eventualmente personalizaremos cómo RSpec interactuará con nuestro código (*spec/spec_helper.rb* y *spec/rails_helper.rb*). Estos dos últimos archivos incluyen muchos comentarios para explicar qué proporciona cada personalización. No necesitas leerlos ahora mismo, pero a medida que RSpec se convierta en una parte habitual de tu kit de herramientas de Rails, te recomiendo encarecidamente que los leas y experimentes con los diferentes ajustes. Esa es la mejor manera de entender qué hacen.

Antes de continuar, echemos un vistazo rápido a algo importante en el archivo *rails_helper.rb* generado. Cerca del final, verás una línea comentada:

spec/rails_helper.rb

```
# config.infer_spec_type_from_file_location!
```

En versiones anteriores de rspec-rails, esta línea estaba *descomentada* de forma predeterminada, lo que significaba que RSpec detectaba automáticamente el tipo de spec (model, request, system, etc.) según la ubicación del archivo en el directorio *spec/*. A partir de rspec-rails 8.0, este comportamiento se considera *legado*. En su lugar, los archivos de spec ahora declaran su tipo de forma explícita, así:

```
RSpec.describe User, type: :model do
  # ...
end
```

Los generadores de RSpec ya incluyen la declaración `type:` en los archivos de spec generados, por lo que todo funciona sin problemas. A lo largo de este libro, verás `type: :model`, `type: :request`, `type: :system`, y otros. Este enfoque explícito deja claro de un vistazo qué tipo de spec estás mirando — sin necesidad de comprobar en qué directorio se encuentra el archivo.

A continuación — y esto es opcional — me gusta cambiar la salida de RSpec del formato predeterminado al formato de *documentación*, que es más fácil de leer. Esto hace que sea más sencillo ver qué specs están pasando y cuáles están fallando mientras se ejecuta tu suite. También proporciona un esquema atractivo de tus specs para — lo adivinaste — fines de documentación. Abre el archivo *.rspec* que se acaba de crear y edítalo para que quede así:

.rspec

```
--require spec_helper
--format documentation
```

Como alternativa, también puedes añadir el flag `--warnings` a este archivo. Cuando las *advertencias* están habilitadas, la salida de RSpec incluirá todas y cada una de las advertencias lanzadas por tu aplicación y las gemas que utiliza. Esto puede ser útil al desarrollar una aplicación real — presta siempre atención a las advertencias de deprecación que arrojan tus pruebas —, pero con el propósito de aprender a hacer pruebas, te recomiendo desactivarla para reducir el ruido en la salida de tus pruebas. Siempre puedes volver a añadirla más adelante.

El binstub de `rspec`

A continuación, instalemos un *binstub* para el ejecutor de pruebas de RSpec, simplemente para ahorrarnos un poco de escritura. ¡Estaremos ejecutando la suite de pruebas muy a menudo! En tu línea de comandos, genera el binstub:

```
bundle binstubs rspec-core
```

Esto creará un ejecutable *rspec* dentro del directorio *bin* de la aplicación. Si por alguna razón no quieres instalar el binstub, puedes saltarte esta sección — solo recuerda usar el comando `bundle exec rspec` en todos los lugares donde yo use `bin/rspec` a lo largo del libro.

¡Pruébalo!

Todavía no tenemos ninguna prueba, pero podemos comprobar si RSpec está correctamente instalado en la aplicación. Ejecútalo usando el binstub que acabamos de crear:

```
$ bin/rspec
```

Si todo está instalado correctamente, deberías ver una salida parecida a esta:

```
No examples found.

Finished in 0.00019 seconds (files took 0.07574 seconds to load)
0 examples, 0 failures
```

Si tu salida es diferente, vuelve atrás y asegúrate de haber seguido los pasos descritos anteriormente.

¿Puedo eliminar mi carpeta *test*? Si estás comenzando una nueva aplicación desde cero, sí. Si llevas un tiempo desarrollando tu aplicación, primero ejecuta `rails test` para verificar que no haya pruebas dentro de ese directorio que quizás quieras portar a RSpec.

Resumen

En este capítulo, añadimos RSpec como dependencia a los entornos de desarrollo y pruebas de la aplicación, y configuramos una base de datos exclusiva para pruebas con la que nuestras pruebas puedan interactuar. También añadimos archivos de configuración predeterminados para RSpec.

¡Ahora estamos listos para escribir algunas pruebas! En el próximo capítulo, comenzaremos a probar la funcionalidad de la aplicación, empezando por su capa de modelo.

Ejercicios

- Si tienes una aplicación Rails existente que necesita cobertura de pruebas, siéntete libre de comenzar a construir una suite de pruebas ahora, siguiendo los pasos que seguimos para instalar y configurar RSpec en TasteDrivenDishes.
- O, ¡intenta crear una nueva aplicación Rails desde cero! No necesita ser sofisticada ni siquiera única. Una simple lista de tareas o un blog siempre es una excelente opción para aprender, o quizás una herramienta para ayudar a llevar un registro de los elementos de tu colección favorita. ¡Sé tan creativo como quieras!

- *O*, ¡quizás tienes ideas para características que añadirías a TasteDrivenDishes! Si ese es el caso, tómate un poco de tiempo para desarrollarlas. No necesitan ser sofisticadas ni bonitas, pero puede que quieras hacer este trabajo en una copia separada del código para evitar conflictos en capítulos futuros.

Cualquiera que sea el camino que elijas, no te preocupes por las pruebas todavía. Si optaste por la ruta de la aplicación Rails existente o la de una aplicación desde cero, instala y configura RSpec en ella ahora. Asegúrate de haber instalado los gems necesarios y haberlos configurado para la aplicación. Verifica que `bin/rspec` se ejecute con éxito.

3. Specs de modelos

Con RSpec instalado correctamente, pongámoslo a trabajar y comencemos a construir una suite de pruebas confiables. Empezaremos con los componentes fundamentales de TasteDrivenDishes: sus modelos.

En este capítulo, completaremos las siguientes tareas:

- Primero crearemos specs de modelos para los modelos existentes.
- Luego, escribiremos pruebas que pasen para las validaciones, los métodos de clase y los métodos de instancia de un modelo, y organizaremos nuestras specs en el proceso.

Crearemos nuestros primeros archivos spec para los modelos existentes de forma manual. Más adelante, al agregar nuevos modelos a la aplicación, los prácticos generadores de RSpec que configuramos en el capítulo 2 generarán archivos de plantilla vacíos por nosotros.

Para seguir el código de este capítulo, comienza con la carpeta *02-setup* del código de ejemplo. Consulta la carpeta *03-models* para ver el código completo de este capítulo. Consulta "Descarga del código de ejemplo" del Capítulo 1 para ver las instrucciones de descarga.

Anatomía de un spec de modelo

Creo que la forma más fácil de aprender a hacer pruebas es a nivel del modelo, porque hacerlo te permite examinar y probar los componentes fundamentales de una aplicación. Un código bien probado a este nivel proporciona una base sólida para un código base general confiable.

Para empezar, un spec de modelo debería incluir pruebas para lo siguiente:

- Cuando se instancia con atributos válidos, un objeto del modelo debería ser válido.
- Los objetos que no cumplen con los requisitos de validación no deberían ser válidos.
- Los métodos de clase y de instancia funcionan como se espera.

Este es un buen momento para ver la estructura básica de un spec de modelo de RSpec. Me resulta útil pensarlos como esquemas individuales. Por ejemplo, veamos los requisitos más simples de nuestro modelo *User*:

```
describe User do
  it "is valid with an email, password, nickname, and API token"
  it "requires a nickname"
  it "requires a unique nickname"
  it "requires an email"
  it "requires a unique email"
  it "requires a password"
  it "requires an API token"
  it "sets a new user's API token"
end
```

Ampliaremos este esquema en unos minutos, pero esto nos da mucho con qué trabajar para empezar. Es un spec sencillo para un modelo ciertamente simple, pero nos lleva a nuestras primeras cuatro buenas prácticas:

- **Describe un conjunto de expectativas**–en este caso, cómo debería ser el modelo User y cómo debería comportarse.
- **Cada ejemplo (una línea que comienza con `it`) espera solo una cosa**. Observa que estoy probando cada validación por separado. De esta manera, si un ejemplo falla, sé que es por esa validación *específica*, y no tengo que revisar a fondo la salida de RSpec en busca de pistas–al menos, no tan profundamente.
- **Cada ejemplo es explícito**. La cadena descriptiva después de `it` es técnicamente opcional en RSpec. Sin embargo, omitirla hace que tus specs sean más difíciles de leer.
- **La descripción de cada ejemplo comienza con un verbo, no con "should"**. Lee las expectativas en voz alta: *User requires an API token*, *User requires an email*, *User sets a new user's API token*. La legibilidad es importante, ¡y es una característica clave de RSpec!

Con estas buenas prácticas en mente, construyamos un spec para el modelo *User*.

Creación de un spec de modelo

En el capítulo 2, configuramos RSpec para que genere automáticamente archivos de plantilla cada vez que agregamos nuevos modelos y controladores a la aplicación. Sin embargo, podemos

invocar los generadores en cualquier momento. Aquí, usaremos uno para generar un archivo inicial para nuestro primer spec de modelo.

Comienza usando el generador *rspec:model* en la línea de comandos:

```
$ bin/rails g rspec:model user
```

RSpec informa la creación del nuevo archivo:

```
rails g rspec:model user
      create  spec/models/user_spec.rb
```

Abramos el nuevo archivo y echemos un vistazo.

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    pending "add some examples to (or delete) #{__FILE__}"
5  end
```

El nuevo archivo nos ofrece un primer vistazo a la sintaxis y las convenciones de RSpec. Primero, hacemos *require* del archivo *rails_helper* en este archivo, y lo haremos en prácticamente todos los archivos de nuestra suite de pruebas. Esto le indica a RSpec que necesitamos que se cargue la aplicación Rails para que luego pueda ejecutar las pruebas contenidas en el archivo. A continuación, usamos el método *describe* para listar un conjunto de cosas que se espera que haga un *modelo* llamado *User*. Observa la declaración `type: :model`–como comentamos en el capítulo anterior, esto le indica a RSpec cómo tratar el spec. Hablaremos más sobre *pending* en el capítulo 12, cuando comencemos a practicar el desarrollo guiado por pruebas. Por ahora, ¿qué ocurre si ejecutamos esto, usando `bin/rspec`?

```
User
  add some examples to (or delete) /path/to/spec/models/user_spec.rb (PENDING: Not yet im\
plemented)

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) User add some examples to (or delete) /path/to/spec/models/user_spec.rb
    # Not yet implemented
    # ./spec/models/user_spec.rb:4

Finished in 0.00058 seconds (files took 0.66131 seconds to load)
1 example, 0 failures, 1 pending
```

> No es necesario usar generadores para crear archivos de spec, ni *ningún* otro archivo en una aplicación Rails, en cualquier caso. Pero son una buena forma de evitar errores tontos causados por erratas.

Vamos a conservar el bloque *describe*, pero reemplazaremos su contenido con el esquema que creamos hace unos minutos:

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    it "is valid with an email, password, nickname, and API token"
5    it "requires a nickname"
6    it "requires a unique nickname"
7    it "requires an email"
8    it "requires a unique email"
9    it "requires a password"
10   it "requires an API token"
11   it "sets a new user's API token"
12 end
```

Vamos a completar los detalles en un momento, pero si ejecutáramos los specs ahora mismo desde la línea de comandos (escribiendo `bin/rspec` en la línea de comandos), la salida sería algo así:

```
User
  is valid with an email, password, nickname, and API token (PENDING: Not yet implemented)
  requires a nickname (PENDING: Not yet implemented)
  requires a unique nickname (PENDING: Not yet implemented)
  requires an email (PENDING: Not yet implemented)
  requires a unique email (PENDING: Not yet implemented)
  requires a password (PENDING: Not yet implemented)
  requires an API token (PENDING: Not yet implemented)
  sets a new user's API token (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) User is valid with an email, password, nickname, and API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:4

  2) User requires a nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:5

  3) User requires a unique nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:6

  4) User requires an email
    # Not yet implemented
    # ./spec/models/user_spec.rb:7

  5) User requires a unique email
    # Not yet implemented
    # ./spec/models/user_spec.rb:8

  6) User requires a password
    # Not yet implemented
    # ./spec/models/user_spec.rb:9

  7) User requires an API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:10

  8) User sets a new user's API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:11


Finished in 0.00086 seconds (files took 0.83556 seconds to load)
8 examples, 0 failures, 8 pending
```

¡Genial! Ocho specs pendientes. RSpec los marca como *pending* porque aún no hemos escrito

ningún código real para realizar las pruebas. Hagamos eso ahora, comenzando con el primer ejemplo.

Las versiones anteriores de Rails requerían que copiaras manualmente la estructura de tu base de datos de desarrollo en tu base de datos de pruebas mediante una tarea `rake`. Sin embargo, ahora Rails se encarga de esto automáticamente cada vez que ejecutas una migración — la mayor parte del tiempo, al menos. Si obtienes un error que indica que faltan migraciones en tu entorno de pruebas, actualízalas ejecutando `bin/rails db:migrate RAILS_ENV=test`.

La sintaxis de RSpec

En 2012, el equipo de RSpec anunció una nueva alternativa preferida al tradicional `should`, añadida en la versión 2.11. Por supuesto, esto ocurrió apenas unos días después de que publiqué la primera versión completa de este libro — ¡a veces es difícil mantenerse al día con estas cosas!

Este nuevo enfoque soluciona algunos problemas técnicos causados por la antigua sintaxis `should`. En lugar de decir que algo `should` o `should_not` coincidir con la salida esperada, usas `expect` de que algo `to` o `not_to` sea algo distinto.

Como ejemplo, veamos esta prueba de muestra, o *expectación*. En este ejemplo, `2 + 1` siempre debería ser igual a `3`, ¿verdad? En la sintaxis antigua de RSpec, esto se escribiría así:

```
it "adds 2 and 1 to make 3" do
  (2 + 1).should eq 3
end
```

La nueva sintaxis pasa el valor de prueba a un método `expect()` y luego encadena un matcher a este:

```
it "adds 2 and 1 to make 3" do
  expect(2 + 1).to eq 3
end
```

Si estás buscando en Google o Stack Overflow ayuda con alguna pregunta sobre RSpec, o si estás trabajando con una aplicación Rails más antigua, es muy probable que encuentres información que usa la sintaxis antigua con `should`. Esta sintaxis todavía funciona técnicamente en las versiones actuales de RSpec, pero recibirás una advertencia de deprecación cuando intentes usarla. *Puedes* configurar RSpec para desactivar estas advertencias, pero en toda honestidad, es mejor que aprendas a usar la sintaxis preferida con `expect()`.

Entonces, ¿cómo luce esa sintaxis en un ejemplo real? Completemos esa primera expectación de nuestro spec para el modelo User:

spec/models/user_spec.rb

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with an email, password, nickname, and API token" do
    user = User.new(
      email: "test@example.com",
      password: "password",
      nickname: "test",
      api_token: "token"
    )

    expect(user).to be_valid
  end

  it "requires a nickname"
  it "requires a unique nickname"
  it "requires an email"
  it "requires a unique email"
  it "requires a password"
  it "requires an API token"
  it "sets a new user's API token"
end
```

Este sencillo ejemplo usa un *matcher* de RSpec llamado `be_valid` para verificar que nuestro modelo sabe cómo debe lucir para ser válido. Preparamos un objeto (en este caso, una instancia nueva pero sin guardar de User llamada `user`) y luego lo pasamos a `expect` para compararlo con el matcher.

Ahora, si ejecutamos `bin/rspec` desde la línea de comandos nuevamente, vemos un ejemplo exitoso:

```
User
  is valid with an email, password, nickname, and API token
  requires a nickname (PENDING: Not yet implemented)
  requires a unique nickname (PENDING: Not yet implemented)
  requires an email (PENDING: Not yet implemented)
  requires a unique email (PENDING: Not yet implemented)
  requires a password (PENDING: Not yet implemented)
  requires an API token (PENDING: Not yet implemented)
  sets a new user's API token (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your suite's status)

  1) User requires a nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:15

  2) User requires a unique nickname
    # Not yet implemented
    # ./spec/models/user_spec.rb:16

  3) User requires an email
    # Not yet implemented
    # ./spec/models/user_spec.rb:17

  4) User requires a unique email
    # Not yet implemented
    # ./spec/models/user_spec.rb:18

  5) User requires a password
    # Not yet implemented
    # ./spec/models/user_spec.rb:19

  6) User requires an API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:20

  7) User sets a new user's API token
    # Not yet implemented
    # ./spec/models/user_spec.rb:21


Finished in 0.02298 seconds (files took 0.56166 seconds to load)
8 examples, 0 failures, 7 pending
```

¡Felicitaciones, has escrito tu primera prueba!

Los matchers son componentes clave en RSpec, así que tomemos un momento para analizar qué implica `be_valid`. Podríamos haber escrito la prueba como

```
expect(user.valid?).to eq true
```

o

```
expect(user.valid?).to be true
```

En estas variaciones, `eq` y `be` también son matchers. Los matchers ayudan a que las pruebas se lean como lenguaje natural, en lugar del estilo `assert` que quizás hayas visto en otros frameworks. `be_valid` tiene algo más en juego: antes de comparar el lado izquierdo del matcher con el derecho, también llama al método `valid?` sobre el usuario que se está probando. Por eso no necesitamos llamar explícitamente a `user.valid?` en la prueba original.

RSpec incluye muchos matchers integrados; exploraremos algunos de ellos a lo largo del libro. ¡Pero por ahora, escribamos algunas pruebas más!

Probando validaciones

Las validaciones son excelentes para familiarizarse con las pruebas automatizadas. Estas pruebas generalmente se pueden escribir en tan solo unas pocas líneas de código. Completemos nuestra spec de validación de `nickname`:

spec/models/user_spec.rb

```
1  it "requires a nickname" do
2    user = User.new(nickname: nil)
3
4    expect(user).to be_invalid
5  end
```

Esta vez, *esperamos* que un nuevo usuario con el atributo `nickname` explícitamente establecido en nil *no* sea válido. En este caso, el matcher `be_invalid` llama a `user.invalid?` antes de comparar los valores esperado y real.

Eso está bien, pero la prueba no le dice al lector *por qué*. Arreglemos eso:

spec/models/user_spec.rb

```
it "requires a nickname" do
  user = User.new(nickname: nil)

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to include("can't be blank")
end
```

Ahora, la prueba aclara la razón por la que el usuario no es válido, asegurando que el mensaje de error coincida con lo que esperaríamos de esta validación. Lo comprobamos usando el matcher `include` de RSpec, que verifica si un valor está contenido dentro de un valor enumerable (en este caso, `errors`). Y cuando ejecutemos RSpec nuevamente, deberíamos tener dos specs que pasen.

Hay un pequeño problema en nuestro enfoque hasta ahora. Tenemos un par de pruebas que pasan, pero nunca las vimos *fallar*. Esto puede ser una señal de advertencia, especialmente al comenzar. Necesitamos asegurarnos de que el código de prueba haga lo que se pretende, lo que también se conoce como *ejercitar el código bajo prueba.*

Hay un par de cosas que podemos hacer para demostrar que no estamos obteniendo falsos positivos. Primero, invirtamos esa expectativa cambiando `to` por `to_not`:

spec/models/user_spec.rb

```
it "requires a nickname" do
  user = User.new(nickname: nil)

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to_not include("can't be blank")
end
```

Y efectivamente, RSpec reporta un fallo:

```
Failures:

  1) User requires a nickname
    Failure/Error: expect(user.errors[:nickname]).to_not include("can't be blank")
      expected ["can't be blank"] not to include "can't be blank"
    # ./spec/models/user_spec.rb:19:in `block (2 levels) in <top (required)>'

Finished in 0.02385 seconds (files took 0.55168 seconds to load)
8 examples, 1 failure, 6 pending

Failed examples:

rspec ./spec/models/user_spec.rb:15 # User requires a nickname
```

RSpec proporciona `to_not` y `not_to` para este tipo de expectativas. Son intercambiables. Uso `to_not` en el libro, ya que es lo que se usa comúnmente en la documentación de RSpec.

También podemos modificar el código de la aplicación para ver cómo afecta a la prueba. Deshaz el cambio que acabamos de hacer en la prueba (cambia `to_not` de vuelta a `to`), luego abre el modelo `User` y comenta temporalmente la validación de `nickname`:

app/models/user.rb

```ruby
require "open-uri"

class User < ApplicationRecord
  include Clearance::User

  has_many :recipes, dependent: :destroy
  has_many :favorites, dependent: :destroy
  has_many :favorite_recipes, through: :favorites, source: :recipe
  has_many :comments, dependent: :destroy
  has_one_attached :avatar

  # validates :nickname, presence: true, uniqueness: true
  validates :api_token, presence: true, uniqueness: true

  before_validation :set_api_token, on: :create
  before_create :set_avatar

  # remainder of file omitted ...
```

Ejecuta los specs de nuevo. Esta vez, deberías ver nuevamente un fallo. Le dijimos a RSpec que un usuario sin nickname debería ser inválido, pero el código de nuestra aplicación no contemplaba eso.

Estas son formas sencillas de verificar que tus tests funcionan como se espera, especialmente a medida que avanzas desde probar validaciones simples hasta lógica más compleja, y estás probando código que ya ha sido escrito. Si no ves ningún cambio en la salida de los tests, es muy probable que el test no esté interactuando realmente con el código, o que el código se comporte de manera distinta a lo que esperas.

Ahora podemos usar el mismo enfoque para probar la validación de `:email`.

spec/models/user_spec.rb

```
1  it "requires an email" do
2    user = User.new(email: nil)
3
4    expect(user).to be_invalid
5    expect(user.errors[:email]).to include("can't be blank")
6  end
```

Puede que estés pensando que estos tests son relativamente sin sentido–¿qué tan difícil puede ser asegurarse de que las validaciones estén incluidas en un modelo? La verdad es que pueden ser más fáciles de omitir de lo que imaginas. Más importante aún, si piensas en qué validaciones debería tener tu modelo *mientras* escribes los tests (idealmente, y con el tiempo, siguiendo un estilo de desarrollo guiado por pruebas (Test-Driven Development)), es más probable que recuerdes incluirlas.

Ampliemos lo que hemos aprendido hasta ahora para escribir un test un poco más complicado–esta vez, para verificar la validación de unicidad en el atributo nickname:

spec/models/user_spec.rb

```
it "requires a unique nickname" do
  User.create(
    nickname: "test",
    email: "test1@example.com",
    password: "password"
  )

  user = User.new(
    nickname: "test"
  )

  expect(user).to be_invalid
  expect(user.errors[:nickname]).to include("has already been taken")
end
```

Fíjate en una diferencia sutil aquí: en este caso, primero persistimos un usuario (llamando a `create` en `User` en lugar de `new`) para construir nuestros datos de prueba, y luego instanciamos un segundo usuario como sujeto de la prueba en sí. Esto, por supuesto, requiere que el primer usuario persistido sea válido (con nickname, email y password) y tenga la misma dirección de email asignada. En el capítulo 4, veremos utilidades para agilizar este proceso. Mientras tanto, ejecuta `bin/rspec` para ver la salida del nuevo test.

Ahora probemos una validación más compleja. Para hacerlo, dejaremos de lado los tests del modelo `User` y nos centraremos en el modelo `Recipe`.

Supongamos que queremos asegurarnos de que los usuarios no puedan darle a dos de sus recetas el mismo nombre–el nombre debe ser único dentro del ámbito de ese usuario. En otras palabras, *yo* no puedo tener dos recetas llamadas *Vegetable Stir Fry*, pero tú y yo podríamos tener *cada uno* nuestra propia receta llamada *Vegetable Stir Fry*. ¿Cómo podrías probar eso?

Como hicimos con los usuarios, empezaremos creando un nuevo archivo spec para el modelo Recipe:

```
$ bin/rails g rspec:model recipe
```

A continuación, añade dos ejemplos al nuevo archivo. Probaremos que un solo usuario no puede tener dos recetas con el mismo nombre, pero dos usuarios distintos sí pueden tener cada uno una receta con el mismo nombre.

spec/models/recipe_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Recipe, type: :model do
  it "does not allow duplicate recipe names per user" do
    user = User.create(
      nickname: "test-user",
      email:      "test-user@example.com",
      password:   "password"
    )

    category = Category.create(name: "Test Category")

    user.recipes.create(
      name: "Test Recipe",
      category: category
    )

    second_recipe = user.recipes.build(
      name: "Test Recipe",
    )

    expect(second_recipe).to_not be_valid
    expect(second_recipe.errors[:name]).to include("has already been taken")
  end

  it "allows two users to share a project name" do
    user = User.create(
      nickname: "test-user",
      email:      "test-user@example.com",
      password:   "password"
    )

    other_user = User.create(
      nickname: "another-test-user",
      email:      "another-test-user@example.com",
      password:   "password"
    )

    category = Category.create(name: "Test Category")

    user.recipes.create(
      name: "Test Recipe",
      category: category
    )

    second_recipe = other_user.recipes.build(
      name: "Test Recipe",
      category: category
    )
```

```
50
51      expect(second_recipe).to be_valid
52    end
53  end
```

Esta vez, dado que los modelos `User` y `Recipe` están acoplados mediante una relación de Active Record, al igual que los modelos `Recipe` y `Category`, necesitamos proporcionar un poco más de información. En el caso del primer ejemplo, tenemos un usuario al que se le asignan ambas recetas. En el segundo, el mismo nombre de receta se asigna a dos recetas únicas que pertenecen a usuarios únicos. Ten en cuenta que, en ambos ejemplos, tenemos que usar `create` con los usuarios, es decir, persistirlos en la base de datos, para poder asignarlos a las recetas que estamos probando. Y aunque no forma parte de lo que estamos probando aquí, para cumplir con el requisito de la aplicación de que cada receta pertenezca a una categoría, también necesitamos crear una categoría en cada uno de los tests.

Dado que el modelo `Recipe` tiene la siguiente validación:

app/models/recipe.rb

```
validates :name, presence: true, uniqueness: { scope: :user_id }
```

Estos nuevos specs pasarán sin problema. No olvides verificar tu trabajo: intenta comentar temporalmente la validación, o cambia los tests para que esperen algo diferente. ¿Fallan ahora?

Por supuesto, las validaciones pueden ser más complejas que simplemente requerir un scope específico. Las tuyas podrían involucrar una expresión regular compleja o un validador personalizado. Adquiere el hábito de probar estas validaciones, no solo los casos exitosos en los que todo es válido, sino también las condiciones de error. Por ejemplo, en los ejemplos que hemos creado hasta ahora, probamos qué sucede cuando un objeto se inicializa con valores `nil`. Si tienes una validación para asegurarte de que un atributo debe ser un número, intenta enviarle una cadena de texto. Si tu validación requiere que una cadena tenga entre cuatro y ocho caracteres, prueba enviándole tres caracteres y luego nueve.

Probando métodos de instancia

Retomemos ahora las pruebas del modelo User. Tenemos los comienzos de una funcionalidad para tratar a los usuarios nuevos de manera diferente a los usuarios que llevan un tiempo en el

sitio. Quizás algún día podríamos limitar la cantidad de recetas o comentarios que un usuario nuevo puede publicar. Por ahora, simplemente mostramos una insignia al atribuir una receta a un usuario nuevo.

Para manejar esto, tenemos este método en la clase User:

app/models/user.rb

```
def new_to_site?
  created_at > 1.month.ago
end
```

Podemos usar las mismas técnicas básicas que utilizamos en nuestros ejemplos de validación para crear un ejemplo exitoso de esta funcionalidad:

spec/models/user_spec.rb

```
1  it "indicates a new user" do
2    user = User.new(created_at: Time.now)
3
4    expect(user.new_to_site?).to be true
5  end
```

Genial, pero ¿qué hay de un usuario que lleva un tiempo aquí? También deberíamos probarlo:

spec/models/user_spec.rb

```
1  it "indicates an established user" do
2    user = User.new(created_at: 1.month.ago)
3
4    expect(user.new_to_site?).to be false
5  end
```

No está mal, pero pulamos un poco estos tests con algo de magia de matchers. ¿Recuerdas que anteriormente en este capítulo usamos `be_valid` y `be_invalid` como matchers para probar la validez de un objeto User? ¡También podemos usar `be_` con nuestros propios métodos que devuelven booleanos!

spec/models/user_spec.rb

```
it "indicates a new user" do
  user = User.new(created_at: Time.now)

  expect(user).to be_new_to_site
end

it "indicates an established user" do
  user = User.new(created_at: 1.month.ago)

  expect(user).to_not be_new_to_site
end
```

Personalmente, me encanta esta funcionalidad de RSpec. Creo que hace que los tests se lean más como documentación y menos como código. Si tú o tu equipo no están de acuerdo, no hay nada de malo en las versiones anteriores de estos tests.

De cualquier manera, estamos estableciendo un patrón para las pruebas: crear datos de prueba y luego indicarle a RSpec cómo esperamos que se comporten. Sigamos adelante.

Probando métodos de clase y scopes

Nuestros usuarios pueden buscar títulos de recetas por un término proporcionado. A modo de demostración, actualmente está implementado como un scope en el modelo Recipe:

app/models/recipe.rb

```
scope :by_word_in_name, ->(query) {
  where("name LIKE ?", "%#{query}%") if query.present?
}
```

Agreguemos otra prueba a `recipe_spec` para cubrir esto:

spec/models/recipe_spec.rb

```
it "finds recipes that contain the search term in their name" do
  user = User.create(
    nickname: "test-user",
    email: "test-user@example.com",
    password: "password"
  )

  category = Category.create(name: "Test Category")

  first_recipe = user.recipes.create(
    name: "Pepperoni Pizza",
    category: category
  )

  second_recipe = user.recipes.create(
    name: "Cheese Pizza",
    category: category
  )

  results = Recipe.by_word_in_name("pepperoni")

  expect(results).to include(first_recipe)
  expect(results).to_not include(second_recipe)
end
```

El scope `by_word_in_name` debería devolver una colección de recetas que coincidan con el término de búsqueda, y esa colección solo debería incluir esas recetas, no las que no contienen el término.

Esta prueba nos da otras cosas con las que experimentar: ¿Qué pasa si invertimos las variaciones `to` y `to_not` en las pruebas? ¿O si añadimos más recetas que contengan el término de búsqueda?

Probando todos los casos

Hemos probado el camino feliz —un usuario busca un término para el cual podemos devolver resultados— pero ¿qué pasa cuando la búsqueda no devuelve ningún resultado? Sería mejor que también lo probáramos. La siguiente spec debería bastar:

spec/models/recipe_spec.rb

```
it "returns an empty collection when no recipes matching the search term are found" do
  user = User.create(
    nickname: "test-user",
    email: "test-user@example.com",
    password: "password"
  )

  category = Category.create(name: "Test Category")

  first_recipe = user.recipes.create(
    name: "Pepperoni Pizza",
    category: category
  )

  second_recipe = user.recipes.create(
    name: "Cheese Pizza",
    category: category
  )

  results = Recipe.by_word_in_name("veggie")

  expect(results).to be_empty
end
```

Esta spec verifica el valor devuelto por `Recipe.by_word_in_name("veggie")`. Como la colección resultante *está* vacía, ¡la spec pasa! No solo estamos probando los resultados ideales, sino también las búsquedas sin resultados.

Más sobre los matchers

Ya hemos visto cuatro matchers en acción: `be_valid`, `eq`, `include` y `be_empty`. Primero usamos `be_valid`, que proviene de la gema *rspec-rails* para probar la validez de un modelo de Rails. `eq` e `include` provienen de *rspec-expectations*, que se instala junto con *rspec-rails* cuando configuramos nuestra app para usar RSpec en el capítulo anterior.

Una lista completa de los matchers predeterminados de RSpec se puede encontrar en el *README* del repositorio rspec-expectations en GitHub. A lo largo de este libro veremos varios de ellos. En el capítulo 9, exploraremos cómo crear nuestros propios matchers personalizados.

Resumen

Este capítulo se centró en las pruebas de modelos, pero hemos cubierto muchas otras técnicas importantes que querrás usar en otros tipos de specs a medida que avances:

- **Usa expectativas activas y explícitas**: Usa verbos para explicar cuáles deberían ser los resultados de un ejemplo. Intenta verificar solo un resultado por ejemplo. (Hablaremos de las excepciones a esto en un capítulo posterior.)
- **Prueba tanto lo que esperas que *suceda* como lo que esperas que *no suceda***: Piensa en ambos caminos al escribir ejemplos y escribe las pruebas en consecuencia.
- **Prueba los casos límite**: Si tienes una validación que requiere que una contraseña tenga entre cuatro y diez caracteres, no te conformes con probar solo una contraseña de ocho caracteres. Un buen conjunto de pruebas verificaría en cuatro y diez caracteres, así como en tres y once. (Por supuesto, también podrías aprovechar la oportunidad para preguntarte por qué permitirías contraseñas tan cortas o no permitirías contraseñas más largas. Las pruebas también son una buena oportunidad para reflexionar sobre los requisitos y el código de una aplicación.)

Con una sólida colección de specs de modelos incorporada en tu app, estás en buen camino hacia un código más confiable. ¡Excelente trabajo!

Ejercicios

Si estás siguiendo el libro con tu propio código Rails sin pruebas, echa un vistazo a sus modelos ahora. ¿Qué atributos se corresponden con sus tablas en la base de datos? ¿Cómo se validan los datos? ¿Qué otra lógica de negocio contienen?

Todos estos son excelentes candidatos para tu primera cobertura de pruebas. Empieza generando una spec de modelo para un modelo dado, luego esboza los escenarios que quieres probar. Piensa en cómo construir solo los datos de prueba necesarios para cada escenario y en cómo demostrarían que tu código se comporta como deseas. Luego escribe cada spec y ejecútala. Te recomiendo hacerlo una prueba a la vez. No te preocupes si te estás repitiendo de prueba en prueba. En el próximo capítulo exploraremos las opciones de deduplicación.

Mientras escribes y ejecutas tus nuevas especificaciones, ¿notas algo inesperado en el código de tu aplicación? Por ejemplo, ¿alguna funcionalidad que no funciona del todo como creías, o código que se beneficiaría de una refactorización? ¡Felicidades! ¡Ya estás viendo los beneficios del diseño de software guiado por pruebas!

Acerca de Left of the Dev

Left of the Dev (anteriormente Everyday Rails) trata sobre el desarrollo de software por diversión, para bien y, a veces, por dinero. Está dirigido a desarrolladores de software pragmáticos que quieren lograr resultados y encontrar las mejores herramientas y técnicas para conseguirlo. Visita Left of the Dev en https://leftofthe.dev/.

Acerca del autor

Aaron Sumner es un ingeniero de software con casi 35 años de experiencia profesional, desarrollando aplicaciones web en las industrias de la educación y la publicación. Desde 2005, ha trabajado principalmente en Ruby on Rails, aunque en su puesto como Engineering Manager en O'Reilly Media, dedica tiempo a Python y Go. Disfruta ayudando a equipos de desarrollo pequeños a alcanzar objetivos ambiciosos mediante el uso pragmático de la tecnología.

Aaron vive en Saint Louis, Misuri, junto a su esposa, Caitlin, y su perro, Lieutenant Dan. Fuera del trabajo, disfruta de la carpintería y de estar al aire libre.

Visita el sitio personal de Aaron en https://aaronsumner.com/.

Colofón

La imagen de portada original “Camión viejo a la luz de la mañana temprana, fotografía de archivo” es obra del colaborador de iStockphoto Habman_18.