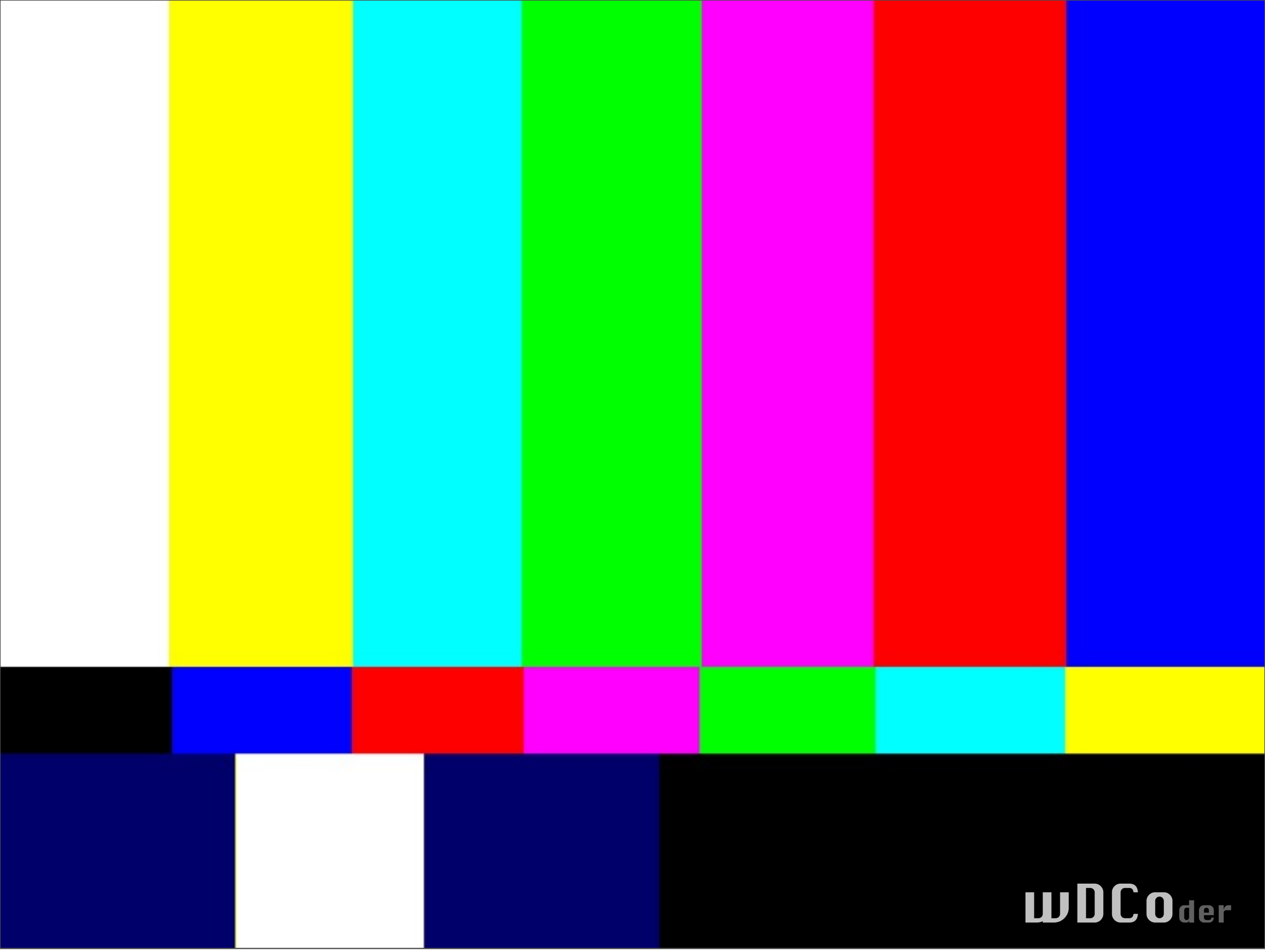
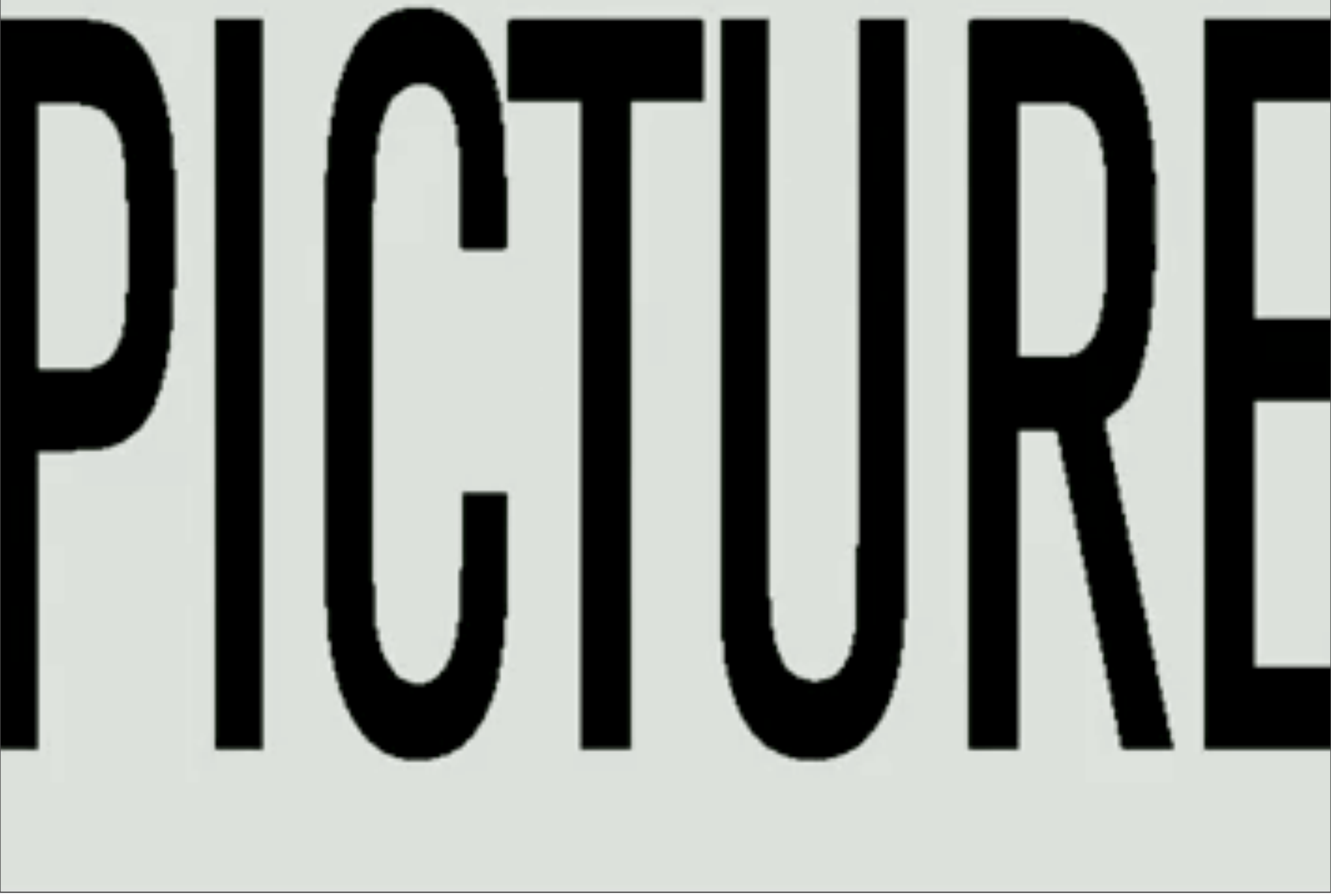

wDCoder

Thank you. Good evening everybody. As John said, my name is Daniel Hedrick. I'd like to thank him for inviting me to speak this evening. [slide]



Thank you. Good evening everybody. As John said, my name is Daniel Hedrick. I'd like to thank him for inviting me to speak this evening. [slide]

Capybara

Daniel Hedrick
@dcoder

https://github.com/dcoder2099/intro_to_capybara

Tonight I'm going to be talking about Capybara. Capybara, or [HydroKOH-erus] [HydroKAY-eris]*, the largest living rodent in the world, and native to South America. It's closest relatives are the Agouti, Chinchilla, Nutria, and Guinea Pig. They can grow 4' in length; 2' in height at the shoulder, and weigh upwards of 150 pounds. What does that have to do with software testing? Nothing... just thought it was interesting.

* Hydrochoerus hydrochaeris

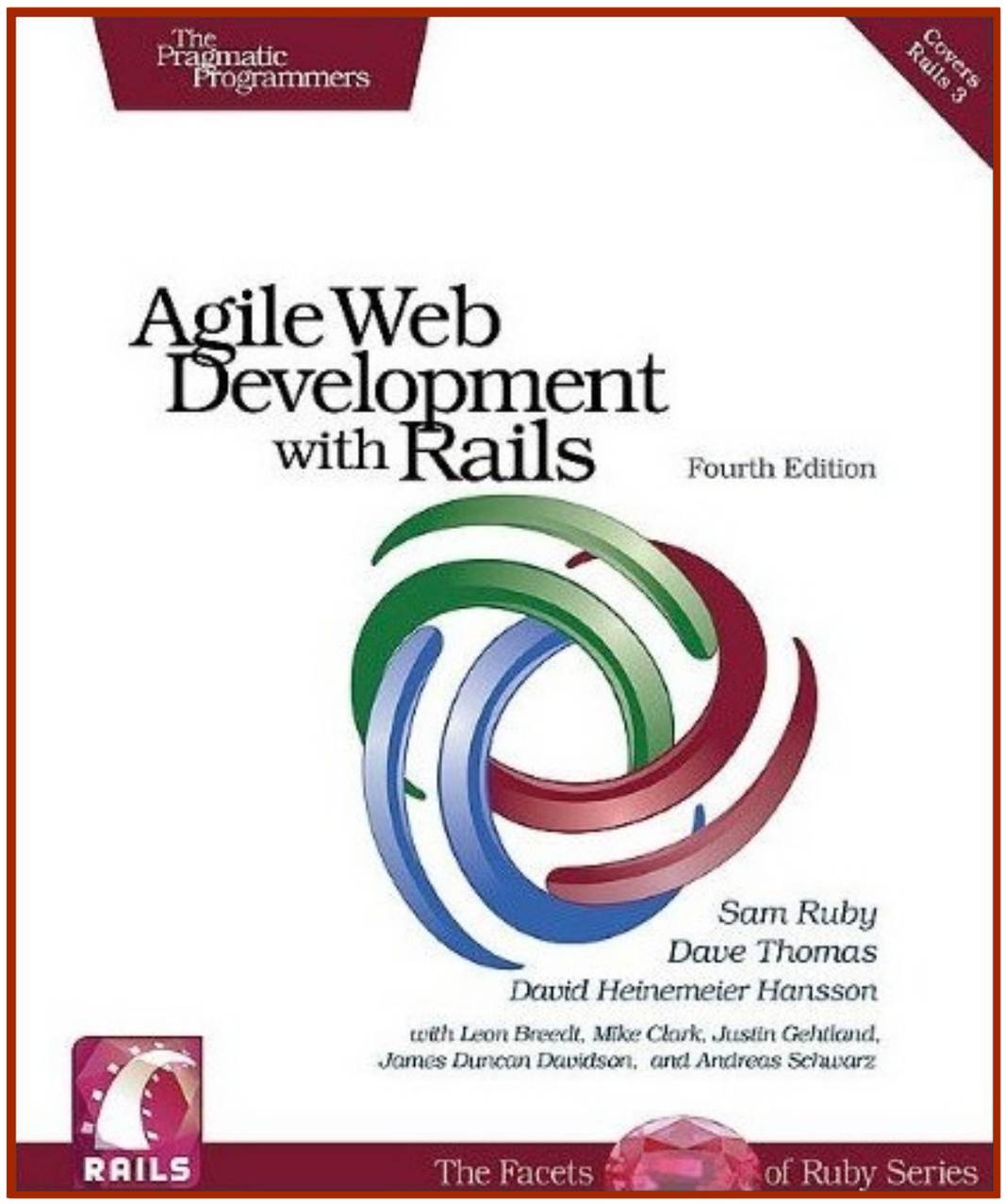
Step 1: Quiz

Let’s begin with a brief quiz...

* How many "rails beginners" do we have here tonight?
* How many of you (all of you) began your learning with the book "Agile Web Development With Rails"?
[txn click]
This book, now in its 4th edition is still considered by many to be the seminal "getting started". I'll be referring to it's Depot application as I discuss Capybara later on.

* How many of you remember working your way through it’s “Depot” web store application?
[slide]

Step 1: Quiz



Let’s begin with a brief quiz...

* How many "rails beginners" do we have here tonight?
* How many of you (all of you) began your learning with the book "Agile Web Development With Rails"?
[txn click]
This book, now in its 4th edition is still considered by many to be the seminal "getting started". I'll be referring to it's Depot application as I discuss Capybara later on.

* How many of you remember working your way through it’s “Depot” web store application?
[slide]

Let's talk briefly about tests in general. Tests are useful. They help validate understanding and they help verify behavior. In the case of well-written tests, they are AWESOME.

But you have to be careful or you can be drawn down a deep rabbit hole.
[rabbit hole]

After nearly 20 years of writing software, and extensive experimentation in developing automated tests, I'll assert that my first key lesson tonight is that your test automation should employ a dual-focus testing strategy. First, build a solid foundation with unit tests. Then, focus on front-end testing. The unit tests ensure each given class's responsibilities are trustworthy. The integration tests then exercise the entire stack from front-end to the database, including any related processing (emails, offline processing).

Solid integration tests must be written that cover all aspects of what you expect the user to be able to do, but there is a very good way to "sneak up" on integration testing and that's by starting out slowly, or actually quickly as the case is.
[slide]

Let's talk briefly about tests in general. Tests are useful. They help validate understanding and they help verify behavior. In the case of well-written tests, they are AWESOME.

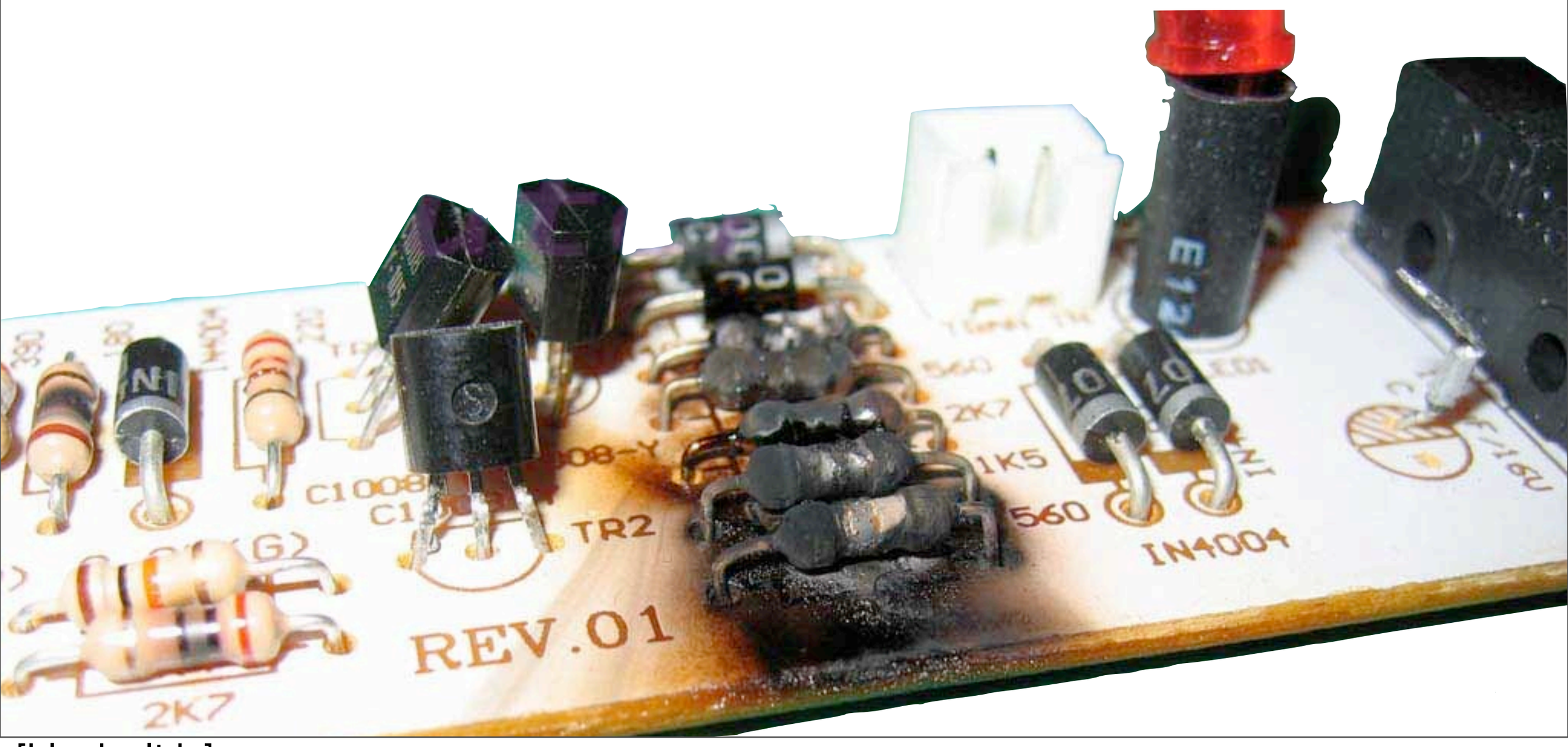
But you have to be careful or you can be drawn down a deep rabbit hole.
[rabbit hole]

After nearly 20 years of writing software, and extensive experimentation in developing automated tests, I'll assert that my first key lesson tonight is that your test automation should employ a dual-focus testing strategy. First, build a solid foundation with unit tests. Then, focus on front-end testing. The unit tests ensure each given class's responsibilities are trustworthy. The integration tests then exercise the entire stack from front-end to the database, including any related processing (emails, offline processing).

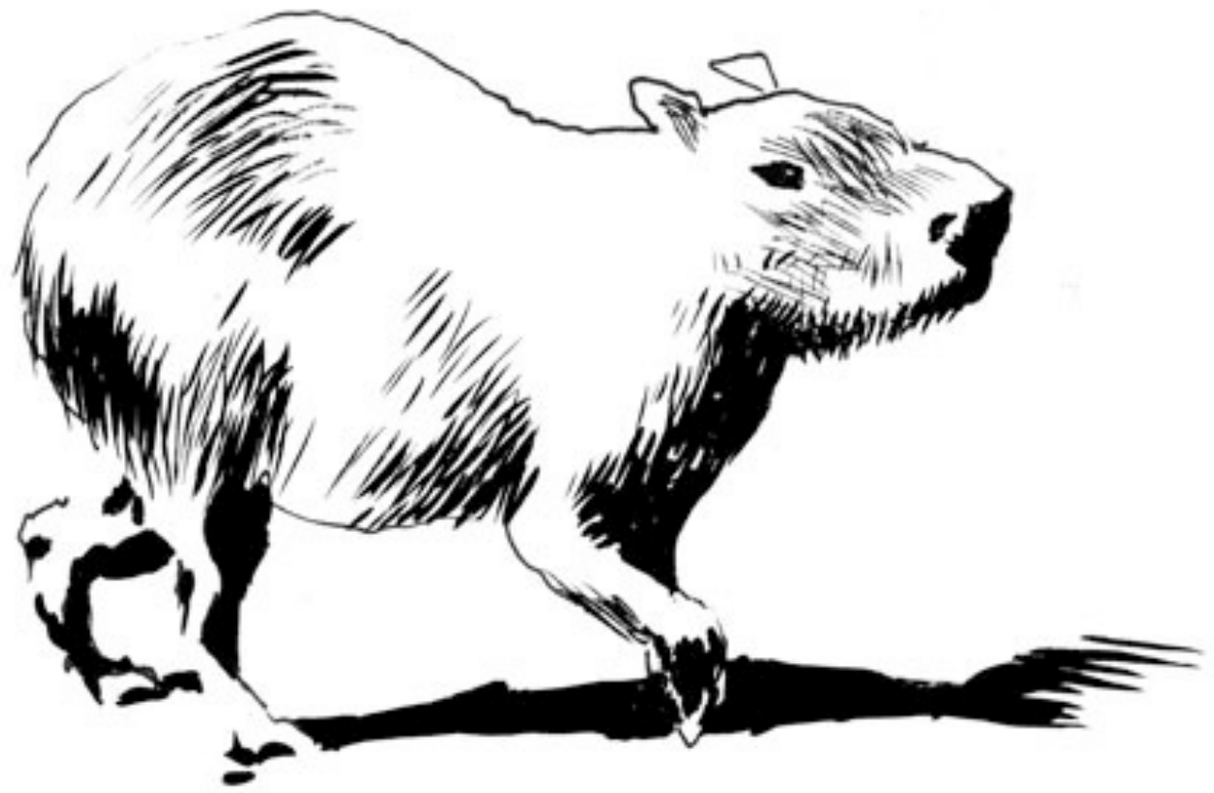
Solid integration tests must be written that cover all aspects of what you expect the user to be able to do, but there is a very good way to "sneak up" on integration testing and that's by starting out slowly, or actually quickly as the case is.
[slide]

[blank slide]
A "Smoke Test" is a run-through of specific, known application behavior that can confirm the application is _basically_ working. In other industries, this sort of test is also known as a "pressure test" or "leak test" and according to Wikipedia, its etymology can be traced back to late 19th century plumbing practices where a closed plumbing loop is filled with smoke while the lines are checked for leaks. A more-likely link to our industry comes from electrical engineering, where an improperly wired circuit, when powered, would cause the board and components to start smoking.
[click]
In any case the idea is that if these tests pass, then the product is basically functional and more thorough regression tests can be run. A failure in a smoke test is a quick indicator that something is amiss.

Smoke if you got 'em



[blank slide]
A "Smoke Test" is a run-through of specific, known application behavior that can confirm the application is _basically_ working. In other industries, this sort of test is also known as a "pressure test" or "leak test" and according to Wikipedia, its etymology can be traced back to late 19th century plumbing practices where a closed plumbing loop is filled with smoke while the lines are checked for leaks. A more-likely link to our industry comes from electrical engineering, where an improperly wired circuit, when powered, would cause the board and components to start smoking.
[click]
In any case the idea is that if these tests pass, then the product is basically functional and more thorough regression tests can be run. A failure in a smoke test is a quick indicator that something is amiss.



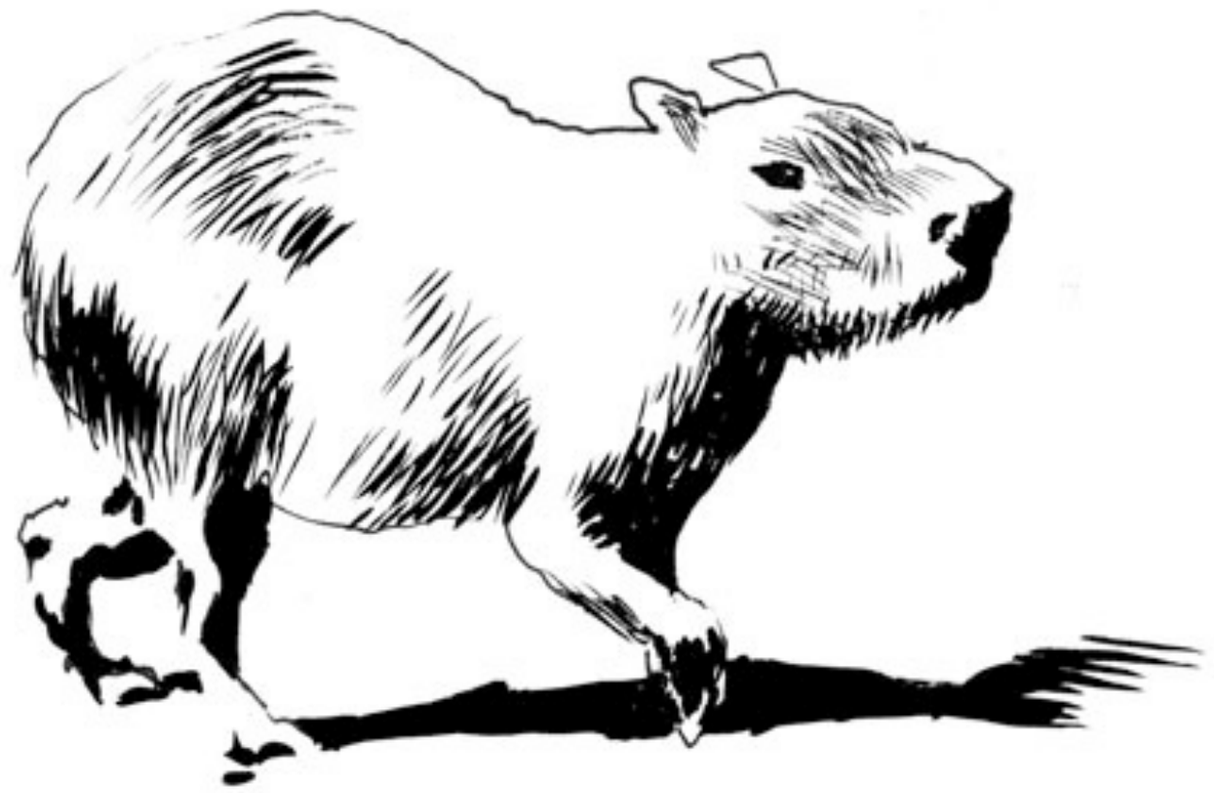
https://github.com/jnicklas/capybara

By employing Capybara in our Rails and RSpec test automation we gain a straightforward and useful DSL for automating these smoke tests.

From Capybara's own README.md:
[click]
"Capybara helps you test web applications by simulating how a real user would interact with your app."

Real user interacting with our app, huh? That sounds exactly like something we could use to drastically improve our verification time of a smoke test, right?

We'll get into more detail about exactly how Capybara does this in just a minute, but first, let’s focus a little on what, exactly, our application does and what, exactly, a smoke test plan _is_.
[slide]



https://github.com/jnicklas/capybara

"Capybara helps you test web applications
by simulating how a real user would
interact with your app."

By employing Capybara in our Rails and RSpec test automation we gain a straightforward and useful DSL for automating these smoke tests.

From Capybara's own README.md:
[click]
"Capybara helps you test web applications by simulating how a real user would interact with your app."

Real user interacting with our app, huh? That sounds exactly like something we could use to drastically improve our verification time of a smoke test, right?

We'll get into more detail about exactly how Capybara does this in just a minute, but first, let's focus a little on what, exactly, our application does and what, exactly, a smoke test plan _is_.
[slide]

The Depot

Remember a few minutes ago, when I asked about the "Agile Web Development With Rails" book? The application the reader builds while working through that book is a rudimentary shopping cart that demonstrates several of the core features of Ruby on Rails, and I could think of no better starting point for this talk.

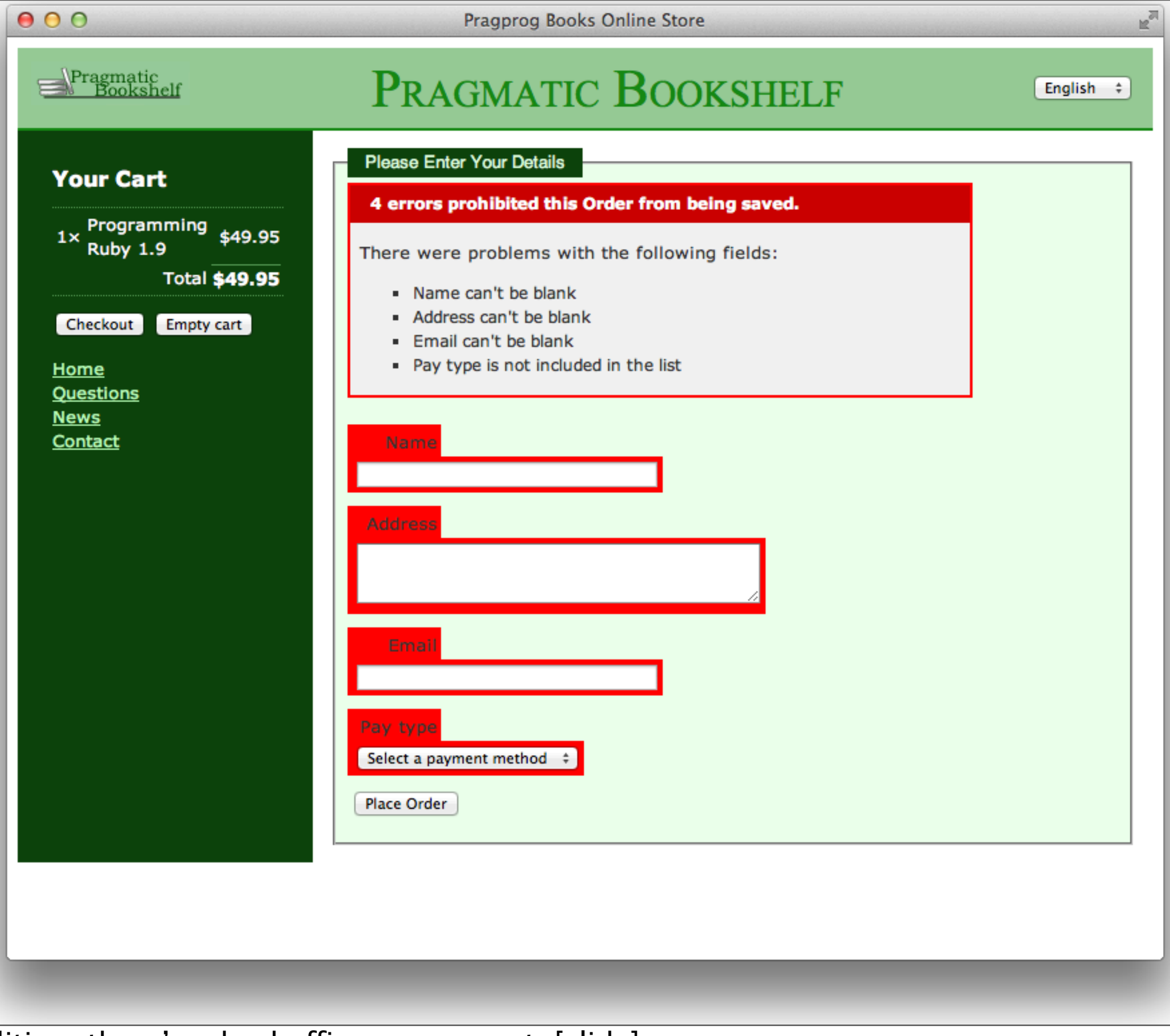
The application is a simple product browser: [click]: [store/pragprog_depot.png]
It manages the cart[slide]

Remember a few minutes ago, when I asked about the "Agile Web Development With Rails" book? The application the reader builds while working through that book is a rudimentary shopping cart that demonstrates several of the core features of Ruby on Rails, and I could think of no better starting point for this talk.

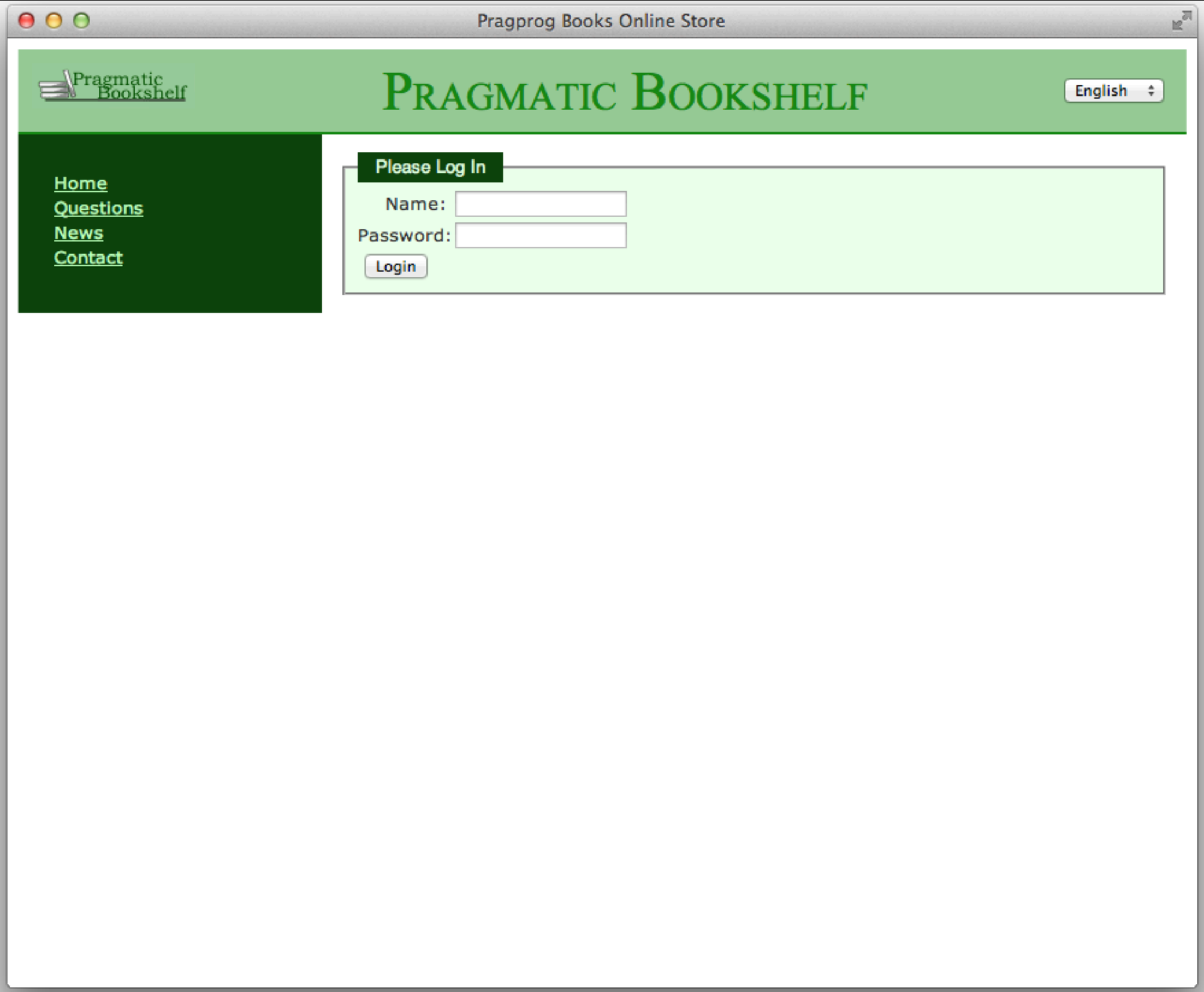
The application is a simple product browser: [click]: [store/pragprog_depot.png]
It manages the cart[slide]

Includes forms. [slide]

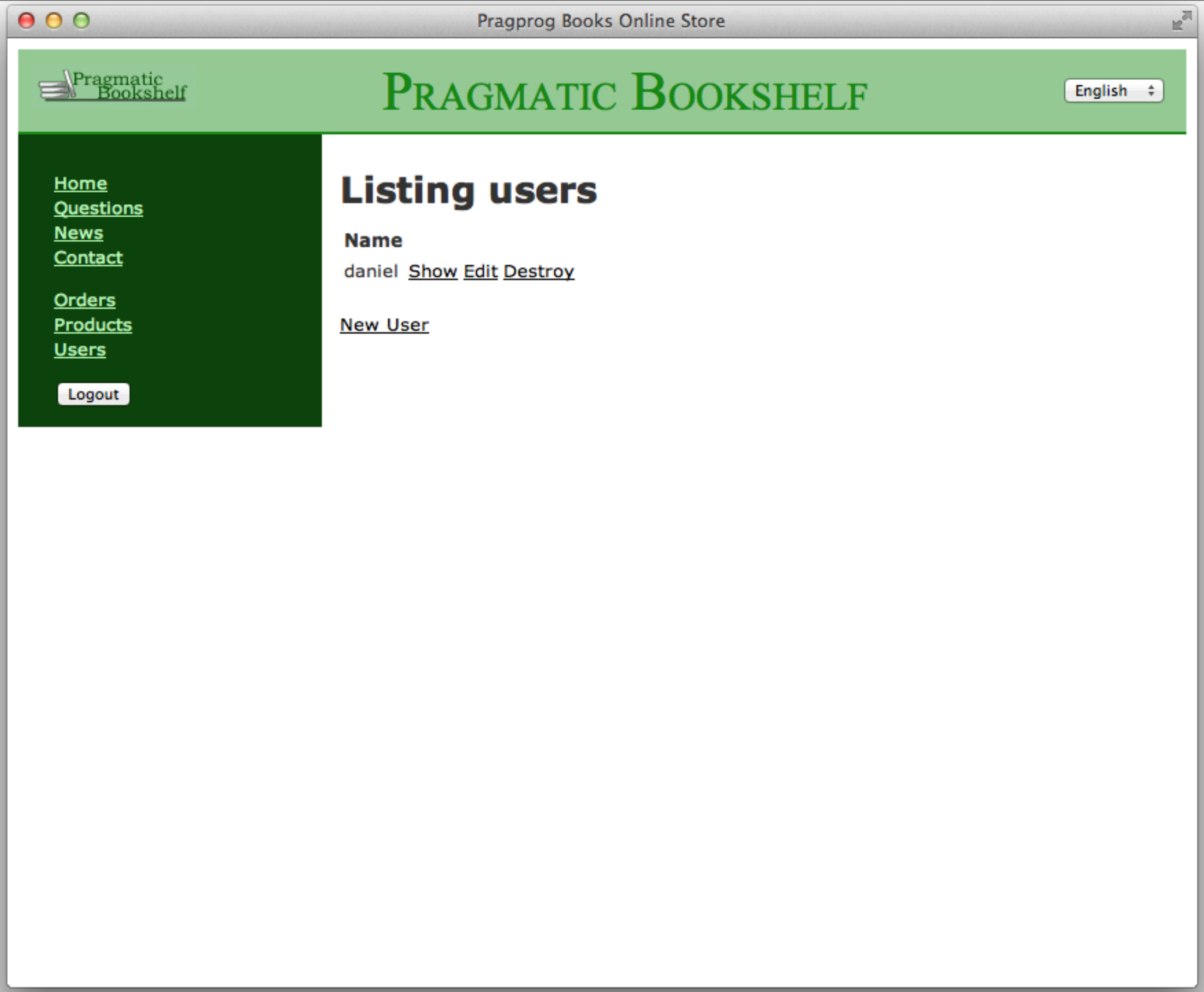
and performs field validation.



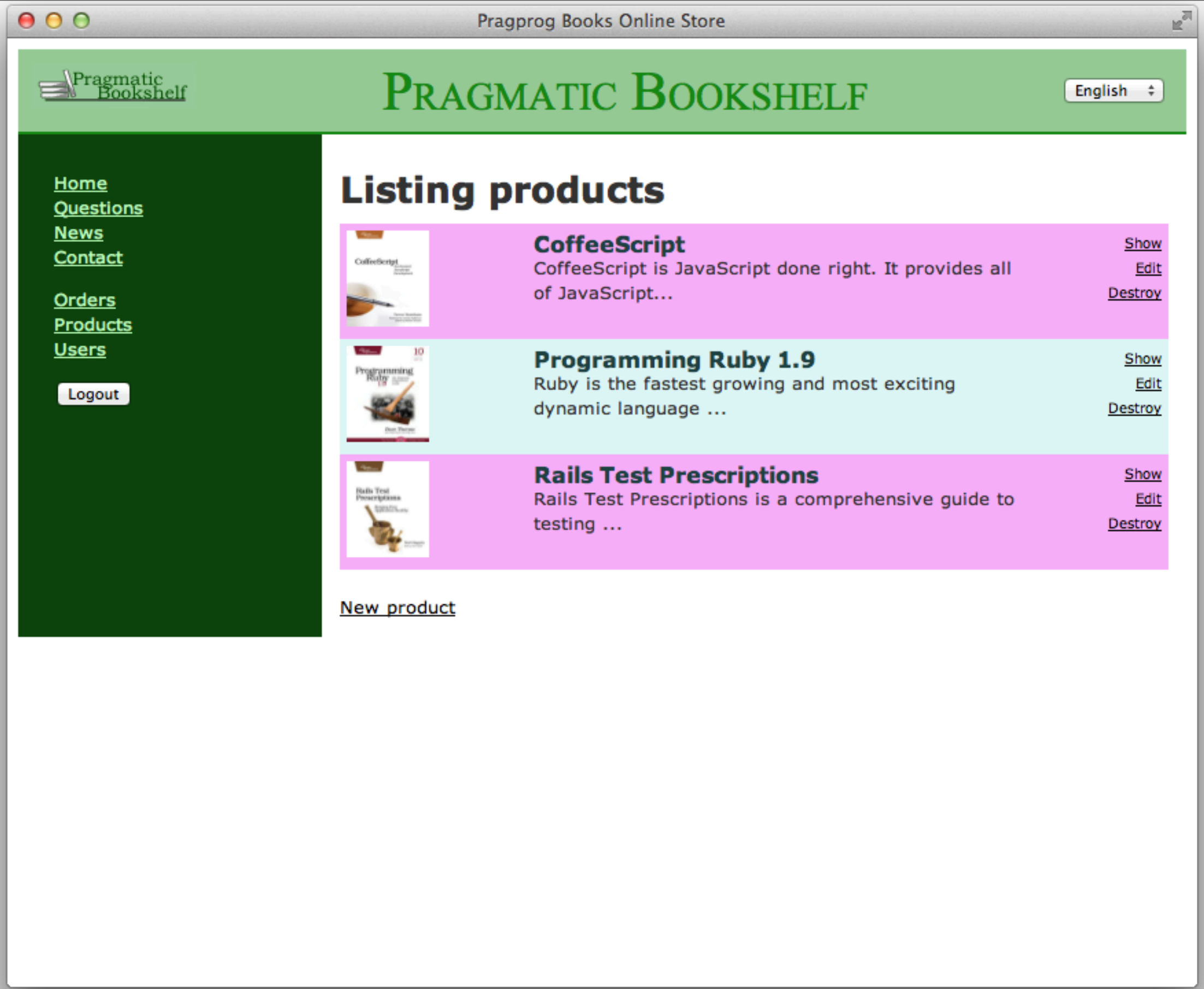
In addition, there's a backoffice component: [slide]



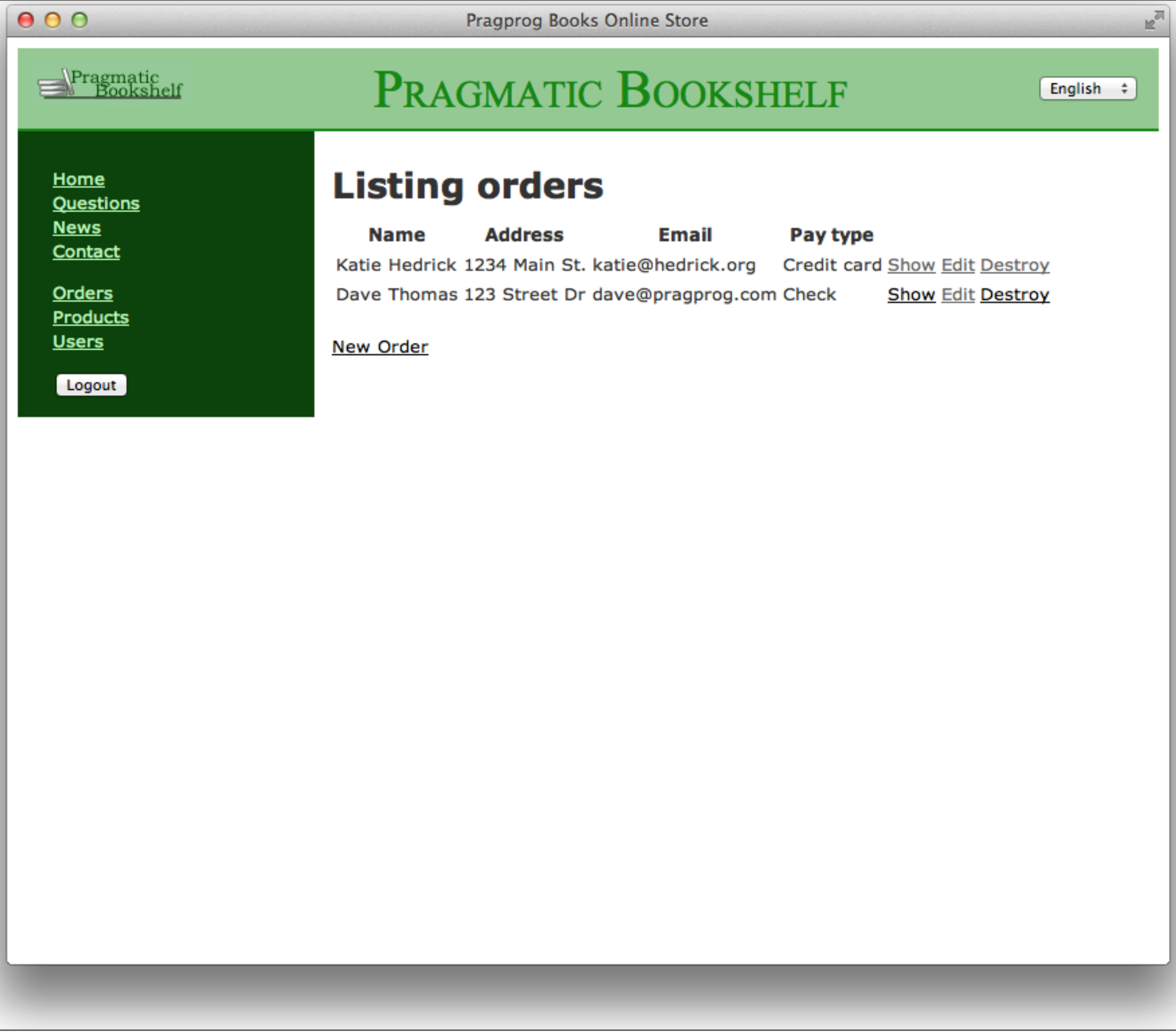
That provides CRUD operations for Users [slide]



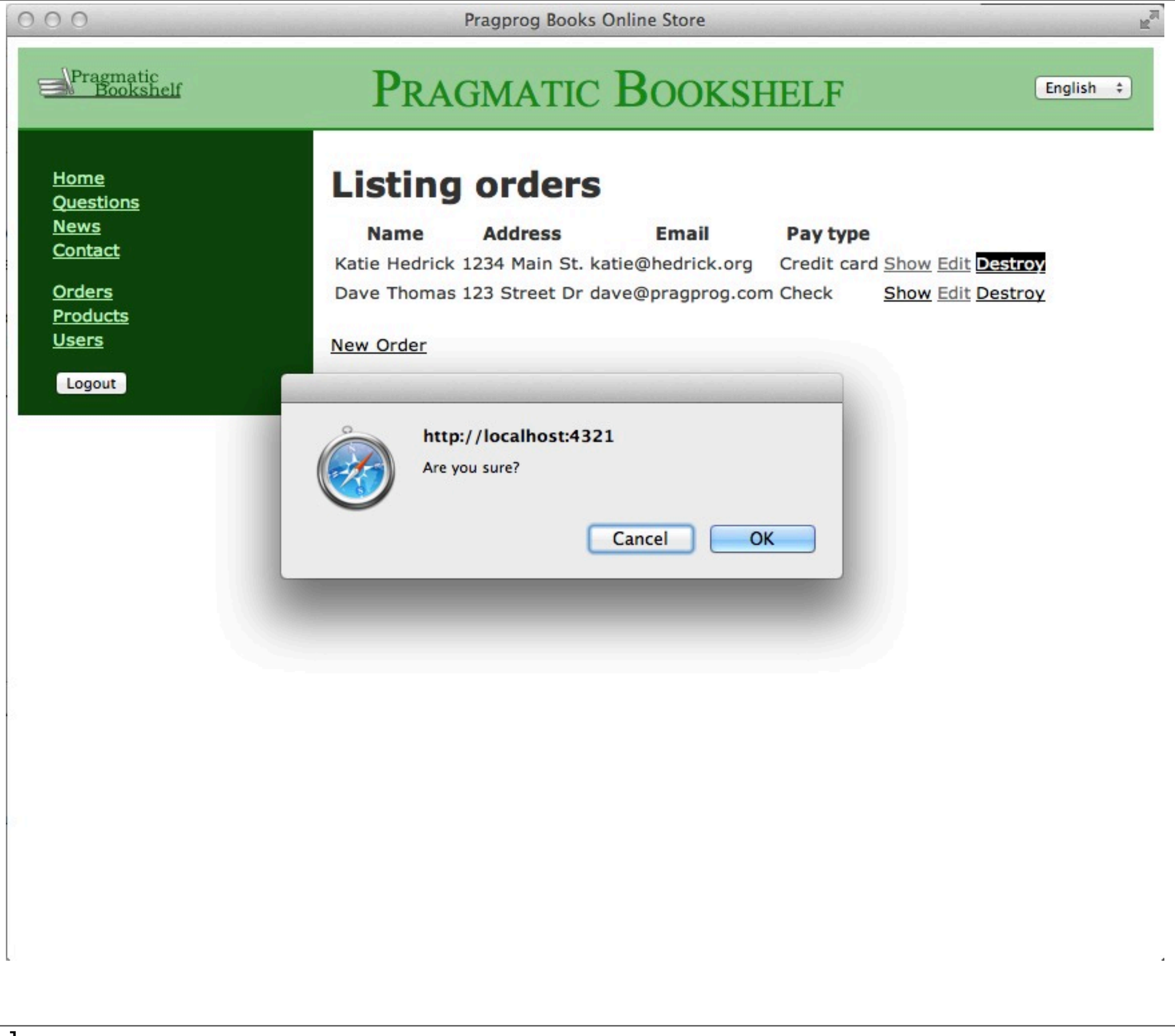
Products [slide]



And Orders.
[slide]



It even has javascript modal dialogs.
[slide]


Pragprog Books Online Store
Pragmatic Bookshelf
PRAGMATIC BOOKSHELF
English
Home
Questions
News
Contact
Orders
Products
Users
Logout
Listing orders
Name Address Email Pay type
Katie Hedrick 1234 Main St. katie@hedrick.org Credit card Show Edit Destroy
Dave Thomas 123 Street Dr dave@pragprog.com Check Show Edit Destroy
New Order
http://localhost:4321
Are you sure?
Cancel
OK

[pause]
[slide]

Smoke Test Plan

Pragmatic Bookshelf
Depot Application

My first task, after I had the cart running, was to walk through the various features of the application and build a "smoke test plan" -- a set of use cases made up of procedural steps that any tester could use to confirm the basic operation of the application.

This is my second key take-away here: Before writing the code for your integration tests, you're welll-served by writing your tests in plain English -- these "test plans" ensure that you know what it is you want to test and how.

Smoke Test Plan

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

Back Office

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

Back Office

- Adding Items to the Cart

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

Back Office

- Adding Items to the Cart
- Emptying the Cart

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

Back Office

- Adding Items to the Cart
- Emptying the Cart
- Checking Out

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

Back Office

- Adding Items to the Cart
- Emptying the Cart
- Checking Out
- Placing an Order

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

- Adding Items to the Cart
- Emptying the Cart
- Checking Out
- Placing an Order

Back Office

- Orders CRUD

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

- Adding Items to the Cart
- Emptying the Cart
- Checking Out
- Placing an Order

Back Office

- Orders CRUD
- Products CRUD

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Smoke Test Plan

The Store

- Adding Items to the Cart
- Emptying the Cart
- Checking Out
- Placing an Order

Back Office

- Orders CRUD
- Products CRUD
- Users CRUD

For the Depot application, I divided the test plan into two major parts:[click] the Store and the Back Office.

Digging deeper, for the Store, I wrote out four specific procedures that sufficiently exercised the front-end allowing us to trust it's operational (remember our definition of smoke test). Those four operations were:

* [click]Adding Items to the Cart
* [click]Emptying the Cart
* [click] Checking Out
* [click] Placing an Order

On the Back Office side, there's the CRUD for Orders [click], Products [click], and Users [click].

For each of these tests, I wrote out a set of steps that any user could follow to help in verifying the smoke test. That means that each test starts "fresh" from the home page and follows all the necessary setup and work to confirm the test.[slide]

Adding Items to the Cart

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.
- Click the "Add to Cart" button for the Programming Ruby 1.9 book

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.

For example, in the test case, "Adding Items to the Cart", the tester should start by:
*[click] visiting the homepage.
*[click] then add one of the items to their cart.
*[click] After confirming that the cart, displayed in the left rail of the application, shows the item they've added...
*[click] the tester adds a different item
*[click] and confirms that it replaces the original item.
[slide]

Emptying the Cart

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Emptying the Cart

- Visit the homepage ("/")

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Emptying the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the Programming Ruby 1.9 book

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Emptying the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Emptying the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.
- Click the "Empty cart" button (left rail)

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Emptying the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.
- Click the "Empty cart" button (left rail)
- Confirm (after a second) that the cart is empty (hidden? zero dollars? no item rows?)

For the test case, "Emptying the Cart", a potentially different tester follows the same steps as the beginning of the first...
*[click] visiting the homepage [click] and adding an item to the cart.
But then, [click] after confirming that it shows up
*[click] this tester clicks the "Empty Cart" button
*[click] and then confirms the cart is truly empty.
[slide]

Adding Items to the Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.

Empty Cart

- Visit the homepage ("/")
- Click the "Add to Cart" button for the Programming Ruby 1.9 book
- Confirm (after a second) that the cart (left rail) refreshes with the Programming Ruby 1.9 book contained within it.
- Click the "Empty cart" button (left rail)
- Confirm (after a second) that the cart is empty (hidden? zero dollars? no item rows?)

Checkout

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.
- Click the "Checkout" button (left rail)
- Confirm the order form is shown.

Place Order

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained within it.
- Click the "Checkout" button (left rail)
- Confirm the order form is shown.
- Fill in the Name, Address, and Email text fields
- Select a Pay type of "Purchase order"
- Click the "Place Order" button
- Confirm returning to the homepage with a message of "Thank you for your order"

The other "Store" tests follow a similar beginning, and then exercise their specific use case. [slide]

Orders

- Visit the homepage ("/")
- Click the "Add to Cart" button for the CoffeeScript book
- Confirm (after a second) that the cart (left rail) refreshes with the CoffeeScript book contained wihtin it.
- Click the "Checkout" button (left rail)
- Fill in the Name, Address, and Email text fields
- Select a Pay type of "Credit card"
- Click the "Place Order" button
- Visit the login screen ("/login")
- Fill in the Name and Password fields
- Click the "Login" button
- Confirm the "Welcome" banner is present
- Click the "Orders" link
- Click the "Show" link for the order
- Confirm the value of the Pay type is "Credit card"
- Click the "Edit" link
- Select a Pay type of "Check"
- Click the "Place Order" button
- Confirm the flash "Order was successfully updated"
- Confirm the value of the Pay type is "Check"
- Click the "Back" link
- Click the "Destroy" link for the order
- Click the "Cancel" button on the popup
- Click the "Orders" link
- Confirm the order is still there
- Click the "Destroy" link for the order
- Click the "Okay" button on the popup
- Confirm the order is gone

Products

- Visit the login screen ("/login")
- Fill in the Name and Password fields
- Click the "Login" button
- Confirm the "Welcome" banner is present
- Click the "Products" link
- Confirm there are three rows in the table
- Click the "New product" link
- Fill in the Title and Description fields
- Click the "Create Product" button
- Confirm the error flash appears with "Image url can't be blank" and "Price is not a number" messages.
- Fill in the Image url and Price fields
- Click the "Create Product" button
- Confirm the flash "Product was successfully created"
- Click the "Edit" link
- Change the Price field
- Click the "Update Product" button
- Confirm the flash "Product was successfully updated"
- Click the "Back" link
- Confirm there are four rows in the table
- Click the "Destroy" link for the last row
- Click the "Okay" button on the popup
- Confirm there are three rows in the table

Users

- Visit the login screen ("/login")
- Fill in the Name and Password fields
- Click the "Login" button
- Confirm the "Welcome" banner is present
- Click the "Users" link
- Click the "New User" link
- Fill in the Name, Password, and Confirm fields
- Click the "Create User" button
- Confirm the flash "User #{name} was successfully created."
- Click the "Logout" button (left rail)
- Visit the login screen ("/login")
- Enter the Name and Password fields
- Click the "Login" button
- Confirm the "Welcome" banner is present
- Click the "Users" link
- Click the "Destroy" link for the user
- Click the "OK" button on the popup
- Confirm the login screen is displayed

The "Back Office" test plans also follow similar procedures with a few added steps, notably logging in, and also editing and deleting items, which follow the basic CRUD operations that these pages provide...
[slide]

[blank slide]
All of these test plans adhere to a rudimentary (and foretelling) syntax:

[click] Visit a page
[click] Click a button or a link
[click] Fill in a form
[click] Confirm behavior or expectation
[slide]

- Visit a page

[blank slide]
All of these test plans adhere to a rudimentary (and foretelling) syntax:

[click] Visit a page
[click] Click a button or a link
[click] Fill in a form
[click] Confirm behavior or expectation
[slide]

- Visit a page
- Click a button or a link

[blank slide]
All of these test plans adhere to a rudimentary (and foretelling) syntax:

[click] Visit a page
[click] Click a button or a link
[click] Fill in a form
[click] Confirm behavior or expectation
[slide]

- Visit a page
- Click a button or a link
- Fill in a form

[blank slide]
All of these test plans adhere to a rudimentary (and foretelling) syntax:

[click] Visit a page
[click] Click a button or a link
[click] Fill in a form
[click] Confirm behavior or expectation
[slide]

- Visit a page
- Click a button or a link
- Fill in a form
- Confirm behavior or expectation

[blank slide]
All of these test plans adhere to a rudimentary (and foretelling) syntax:

[click] Visit a page
[click] Click a button or a link
[click] Fill in a form
[click] Confirm behavior or expectation
[slide]

[blank slide]
So, after all this talk, let's start looking at some code. I've focused quite a bit on behavior and procedure and a lot of "meta" work about testing, but all-in-all, that's really the core value of Capybara. Given a solid test plan, setting Capybara up and writing the smoke tests we've documented is actually quite straightforward.

We start by adding Capybara to our Gemfile.
[click] In the case of the Depot app, I also needed to install RSpec since the book (and code) uses Rails' testing framework.

As a minor bit of housekeeping for this project, [click] I also ran the "rspec:install" Rails generator, as well as added a symlink from "spec/fixtures" to the "test/fixtures" directory, so that our specs could access the same fixtures as the existing tests.

If you look through the GitHub commit history, you'll also see an upgrade to the latest version of rails (3.2.11) and a few minor configuration tweaks to eliminate some console warnings.
[beat, slide]

```
13 # RSpec
14 group :test, :development do
15   gem "rspec-rails", "~> 2.0"
16   gem "capybara"
17 end
```

[blank slide]
So, after all this talk, let's start looking at some code. I've focused quite a bit on behavior and procedure and a lot of "meta" work about testing, but all-in-all, that's really the core value of Capybara. Given a solid test plan, setting Capybara up and writing the smoke tests we've documented is actually quite straightforward.

We start by adding Capybara to our Gemfile.
[click] In the case of the Depot app, I also needed to install RSpec since the book (and code) uses Rails' testing framework.

As a minor bit of housekeeping for this project, [click] I also ran the "rspec:install" Rails generator, as well as added a symlink from "spec/fixtures" to the "test/fixtures" directory, so that our specs could access the same fixtures as the existing tests.

If you look through the GitHub commit history, you'll also see an upgrade to the latest version of rails (3.2.11) and a few minor configuration tweaks to eliminate some console warnings.
[beat, slide]

```
13 # RSpec
14 group :test, :development do
15   gem "rspec-rails", "~> 2.0"
16   gem "capybara"
17 end

~/Development/pragprog_book_depot [master_Δ_]
$ rails generate rspec:install
      create  .rspec
      create  spec
      create  spec/spec_helper.rb

~/Development/pragprog_book_depot [master_Δ_]
$ cd spec/

~/Development/pragprog_book_depot/spec [master_Δ_]
$ ls -alt
total 16
drwxr-xr-x   5 daniel  staff   170 Jan 19 00:42 .
lrwxr-xr-x   1 daniel  staff    16 Jan 19 00:42 fixtures -> ../test/fixtures
drwxr-xr-x   3 daniel  staff   102 Jan 19 00:40 features
drwxr-xr-x@ 26 daniel  staff   884 Jan 19 00:32 ..
-rw-r--r--   1 daniel  staff  1472 Jan 19 00:14 spec_helper.rb

~/Development/pragprog_book_depot/spec [master_Δ_]
$ _
```

[blank slide]
So, after all this talk, let's start looking at some code. I've focused quite a bit on behavior and procedure and a lot of "meta" work about testing, but all-in-all, that's really the core value of Capybara. Given a solid test plan, setting Capybara up and writing the smoke tests we've documented is actually quite straightforward.

We start by adding Capybara to our Gemfile.
[click] In the case of the Depot app, I also needed to install RSpec since the book (and code) uses Rails' testing framework.

As a minor bit of housekeeping for this project, [click] I also ran the "rspec:install" Rails generator, as well as added a symlink from "spec/fixtures" to the "test/fixtures" directory, so that our specs could access the same fixtures as the existing tests.

If you look through the GitHub commit history, you'll also see an upgrade to the latest version of rails (3.2.11) and a few minor configuration tweaks to eliminate some console warnings.
[beat, slide]

```ruby
require 'spec_helper'

describe 'user stories', type: :feature do
  fixtures :products

  it 'buys a product' do
    LineItem.delete_all
    Order.delete_all
    ruby_book = products(:ruby)

    visit "/"
    page.should have_content 'Your Pragmatic Catalog'

    find("#product_#{ruby_book.id}").click

    find('div#cart').should have_content 'Your Cart'
    find('div#cart').should have_content ruby_book.title

    find('div#cart input[value="Checkout"]').click
    page.should have_content 'Please Enter Your Details'

    fill_in 'Name', with: 'Dave Thomas'
    fill_in 'Address', with: '123 The Street'
    fill_in 'Email', with: 'dave@example.com'
    select 'Check', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    orders = Order.all
    orders.size.should eq(1)
    order = orders[0]
    order.name.should eq('Dave Thomas')
    order.address.should eq('123 The Street')
    order.email.should eq('dave@example.com')
    order.pay_type.should eq('Check')

    order.line_items.size.should eq(1)
    line_item = order.line_items[0]
    line_item.product.should eq(ruby_book)

    mail = ActionMailer::Base.deliveries.last
    mail.to.should include('dave@example.com')
    mail[:from].value.should eq('Sam Ruby <depot@example.com>')
    mail.subject.should eq('Pragmatic Store Order Confirmation')
  end
end
```

RSpec + Capybara

```ruby
require 'test_helper'

class UserStoriesTest < ActionDispatch::IntegrationTest
  fixtures :products

  test "buying a product" do
    LineItem.delete_all
    Order.delete_all
    ruby_book = products(:ruby)

    get "/"
    assert_response :success
    assert_template "index"

    xml_http_request :post, '/line_items', product_id: ruby_book.id
    assert_response :success

    cart = Cart.find(session[:cart_id])
    assert_equal 1, cart.line_items.size
    assert_equal ruby_book, cart.line_items[0].product

    get "/orders/new"
    assert_response :success
    assert_template "new"

    post_via_redirect "/orders",
                      order: { name:     "Dave Thomas",
                               address:  "123 The Street",
                               email:    "dave@example.com",
                               pay_type: "Check" }
    assert_response :success
    assert_template "index"
    cart = Cart.find(session[:cart_id])
    assert_equal 0, cart.line_items.size

    orders = Order.all
    assert_equal 1, orders.size
    order = orders[0]

    assert_equal "Dave Thomas",      order.name
    assert_equal "123 The Street",   order.address
    assert_equal "dave@example.com", order.email
    assert_equal "Check",            order.pay_type

    assert_equal 1, order.line_items.size
    line_item = order.line_items[0]
    assert_equal ruby_book, line_item.product

    mail = ActionMailer::Base.deliveries.last
    assert_equal ["dave@example.com"], mail.to
    assert_equal 'Sam Ruby <depot@example.com>', mail[:from].value
    assert_equal "Pragmatic Store Order Confirmation", mail.subject
  end
end
```

ActionDispatch::IntegrationTest

Speaking of the existing tests, as a side note, I converted the existing integration test "test/integration/user_stories_test.rb", seen here on the right, to an RSpec+Capybara test, here on the left, just to ensure everything was working properly.

I won't detail that conversion, but it's also documented in the git history for the project on GitHub.

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

To exercise the four tests of our Store test plan, I wrote a spec, "spec/features/lets_go_shopping_spec.rb" that will enumerate those four tests.

[beat, slide]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

We begin by describing this spec as a :feature,
[pause, slide]

Let's Go Shopping!

```
require 'spec_helper'

describe "let's go shopping", type: :feature do
  fixtures :products
  it 'adds an item to the cart' do
    cs_book = products :coffee
    ruby_book = products :ruby

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart').should have_content ruby_book.price
  end

  it 'empties the cart' do
    ruby_book = products :ruby

    visit '/'
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart input[value="Empty cart"]').click
    find('div#cart').should_not have_content ruby_book.title
```

then load the fixtures for products
[pause, slide]

Let's Go Shopping!

```ruby
require 'spec_helper'

describe "let's go shopping", type: :feature do
  fixtures :products
  it 'adds an item to the cart' do
    cs_book = products :coffee
    ruby_book = products :ruby

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart').should have_content ruby_book.price
  end

  it 'empties the cart' do
    ruby_book = products :ruby

    visit '/'
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart input[value="Empty cart"]').click
    find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby

       visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```ruby
require 'spec_helper'

describe "let's go shopping", type: :feature do
  fixtures :products
  it 'adds an item to the cart' do
    cs_book = products :coffee
    ruby_book = products :ruby

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart').should have_content ruby_book.price
  end

  it 'empties the cart' do
    ruby_book = products :ruby

    visit '/'
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart input[value="Empty cart"]').click
    find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```
1  require 'spec_helper'
2
3  describe "let's go shopping", type: :feature do
4    fixtures :products
5    it 'adds an item to the cart' do
6      cs_book = products :coffee
7      ruby_book = products :ruby
8
9      visit '/'
10     find("#product_#{cs_book.id}").click
11     find('div#cart').should have_content cs_book.title
12     find("#product_#{ruby_book.id}").click
13     find('div#cart').should have_content ruby_book.title
14     find('div#cart').should have_content ruby_book.price
15   end
16
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
```

and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]

Let's Go Shopping!

```ruby
require 'spec_helper'

describe "let's go shopping", type: :feature do
  fixtures :products
  it 'adds an item to the cart' do
    cs_book = products :coffee
    ruby_book = products :ruby

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart').should have_content ruby_book.price
  end

  it 'empties the cart' do
    ruby_book = products :ruby

    visit '/'
    find("#product_#{ruby_book.id}").click
    find('div#cart').should have_content ruby_book.title
    find('div#cart input[value="Empty cart"]').click
    find('div#cart').should_not have_content ruby_book.title
```

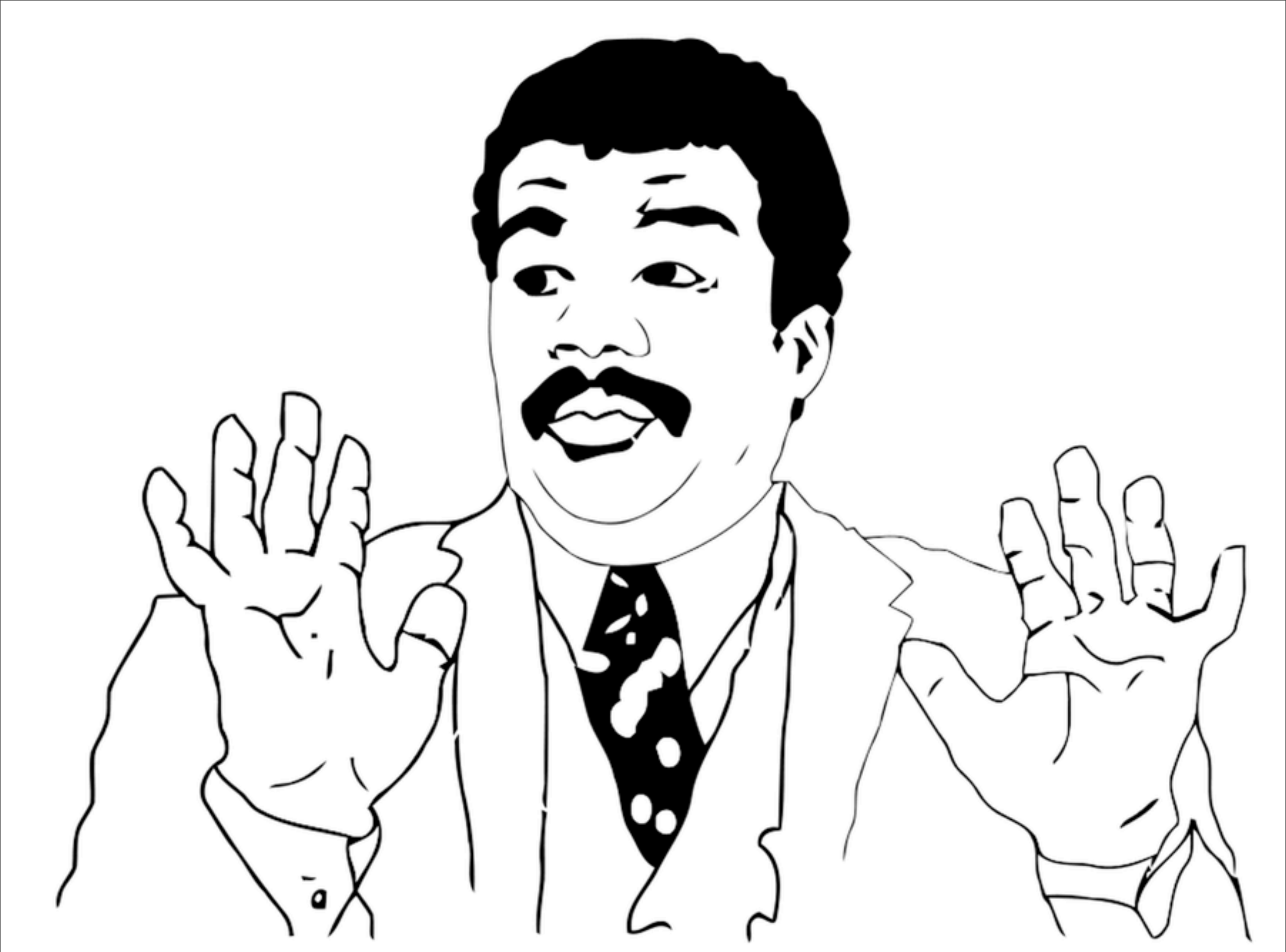
and begin our first test, "adds an item to the cart".
This test begins with some setup of local variables, [click]cs_book and [click]ruby_book, which we load from the fixture, then we employ the Capybara DSL and describe the procedure... almost exactly the same way that our test plan describes.

[click]The "visit" method tells Capybara to access the route described. Here we use a string, but you can also use the equivalent rails routing path, for example "store_path".

[click]Then, using other simple functions of the DSL, we find a specific DOM element on the page (in this case, an input tag with the id of the CoffeeScript book from our fixture and we click it. That click adds the item to the cart and then renders that cart in the left rail of the Depot.

[click]We use a standard RSpec "should" assertion to ensure that the title of the item in the cart is the same as the title we clicked.

[click]The code then repeats the find/click for the Ruby book and asserts its [click] title and price are reflected in the cart.[click]



So that's it. That's the Capybara DSL. Visit. Find. Click. Pretty simple, right? The other three tests are more of the same. In fact, they're all almost exactly the same. Let's look at a few of the differences...
[slide]

Let's Go Shopping!

```
17   it 'empties the cart' do
18     ruby_book = products :ruby
19
20     visit '/'
21     find("#product_#{ruby_book.id}").click
22     find('div#cart').should have_content ruby_book.title
23     find('div#cart input[value="Empty cart"]').click
24     find('div#cart').should_not have_content ruby_book.title
25     find('div#cart').should_not have_content ruby_book.price
26   end
```

To empty the cart, or to proceed to the checkout, we need to click some buttons. Well, we've already clicked one button, right? Except that one was easy to identify with a specific DOM id. These others just require a slightly different selector syntax that uses an attribute bracket, which you can see here on line 23.
[slide]

Let's Go Shopping!

```
38   it 'can place an order' do
39     cs_book = products :coffee
40
41     visit '/'
42     find("#product_#{cs_book.id}").click
43     find('div#cart').should have_content cs_book.title
44     find('div#cart input[value="Checkout"]').click
45     find('div#main').should have_content 'Please Enter Your Details'
46     fill_in 'Name', with: 'Daniel Hedrick'
47     fill_in 'Address', with: '1234 Main Street'
48     fill_in 'Email', with: 'daniel@hedrick.org'
49     select 'Check', from: 'Pay type'
50     find('input[value="Place Order"]').click
51     page.should have_content 'Order Placed'
52   end
```

To actually place an order, we need to fill out the form with our purchase detail. It turns out, that's also pretty simple. The "fill_in" method, seen on lines 46, 47, and 48, is pretty smart in identifying labels, elements and input tags in a way that makes our job easy (at least for this smoke test). Simply tell Capybara to "fill_in" a specific field (eg "Name") with a specific value (eg "Daniel Hedrick") and continue on our merry way. The same with the drop-down, we use the "select" method to tell the control "Pay type" to select "Check".

The final line of our "can place an order" spec is an assertion via flash message that the order has been placed: page.should have_content 'Order Placed'.

We run these tests and...[slide]

Let's Go Shopping!

```
~/Development/pragprog_book_depot [(no branch)_Δ_]
$ brake spec
/Users/daniel/.rvm/rubies/ruby-1.9.3-p194/bin/ruby -S rspec ./spec/features/lets_go_shopping_spec.rb ./spe
No DRb server is running. Running in local process instead ...

let's go shopping
  adds an item to the cart
  can place an order (FAILED - 1)
  can check out
  empties the cart

user stories
  buys a product

Failures:

  1) let's go shopping can place an order
     Failure/Error: page.should have_content 'Order Placed'
       expected there to be text "Order Placed" in "EnglishEspañol Pragmatic Bookshelf Home Questions News
of JavaScript's functionality wrapped in a cleaner, more succinct syntax. In the first book on this exciti
e writing clearer, cleaner, and safer code. $36.00 MyString MyText $9.99 MyString MyText $9.99 Programming
ered fast, you should add Ruby to your toolbox. $49.50 Rails Test Prescriptions Rails Test Prescriptions i
why to test) and from a practical perspective (how to test effectively). It covers the core Rails testing
and Rcov. $34.95"
     # ./spec/features/lets_go_shopping_spec.rb:51:in `block (2 levels) in <top (required)>'

Finished in 1.14 seconds
5 examples, 1 failure
```

it fails.

It turns out that the actual flash message is "Thank you for your order" and after a brief conversation with our hypothetical product manager, we determine that the application's flash message is correct, and the test's (and transitively the test plan's) message is inaccurate.

[slide]

Let's Go Shopping!

```
38  it 'can place an order' do
39    cs_book = products :coffee
40
41    visit '/'
42    find("#product_#{cs_book.id}").click
43    find('div#cart').should have_content cs_book.title
44    find('div#cart input[value="Checkout"]').click
45    find('div#main').should have_content 'Please Enter Your Details'
46    fill_in 'Name', with: 'Daniel Hedrick'
47    fill_in 'Address', with: '1234 Main Street'
48    fill_in 'Email', with: 'daniel@hedrick.org'
49    select 'Check', from: 'Pay type'
50    find('input[value="Place Order"]').click
51    page.should have_content 'Thank you for your order'
52  end
```

The updated assertion reads "page should have content 'Thank you for your order' seen here on line 51.

[click]
And we also update the test plan as this diff shows, some might say, just to be pedantic.

It's at this point, after looking through all the test plans and seeing the redundant code setting up a cart, that we can simplify all four of the smoke test procedures for the store into a single, comprehensive test. Remember, the purpose of a smoke test is to confirm basic functionality and to do it as quickly as possible so that we can pass the build off to our regression testers.

Let's Go Shopping!

```
38   it 'can place an order' do
39     cs_book = products :coffee
40
41     visit '/'
42     find("#product_#{cs_book.id}").click
43     find('div#cart').should have_content cs_book.title
44     find('div#cart input[value="Checkout"]').click
45     find('div#main').should have_content 'Please Enter Your Details'
46     fill_in 'Name', with: 'Daniel Hedrick'
47     fill_in 'Address', with: '1234 Main Street'
48     fill_in 'Email', with: 'daniel@hedrick.org'
49     select 'Check', from: 'Pay type'
50     find('input[value="Place Order"]').click
51     page.should have_content 'Thank you for your order'
52   end
```

3 SMOKE_TEST.md — View file @ 7d07204

```
...  ...  @@ -44,7 +44,8 @@
44   44    1. Fill in the Name, Address, and Email text fields
45   45    1. Select a Pay type of "Purchase order"
46   46    1. Click the "Place Order" button
47        -1. Confirm returning to the homepage with a message of "Order Placed"
     47    +1. Confirm returning to the homepage with a message of "Thank you for
     48    +   your order"
48   49
49   50    ### Backoffice
50   51
```

The updated assertion reads "page should have content 'Thank you for your order' seen here on line 51.

[click]
And we also update the test plan as this diff shows, some might say, just to be pedantic.

It's at this point, after looking through all the test plans and seeing the redundant code setting up a cart, that we can simplify all four of the smoke test procedures for the store into a single, comprehensive test. Remember, the purpose of a smoke test is to confirm basic functionality and to do it as quickly as possible so that we can pass the build off to our regression testers.

Put it altogether...

- Visit the homepage ("/").
- Click the "Add to Cart" button for the CoffeeScript book.
- Confirm that the cart contains the CoffeeScript book.
- Click the "Add to Cart" button for the Programming Ruby 1.9 book.
- Confirm that the cart contains the Programming Ruby 1.9 book.
- Click the "Empty cart" button.
- Confirm that the cart is empty.
- Click the "Add to Cart" button for the Rails Test Prescriptions book.
- Confirm that the cart contains the Rails Test Prescriptions book.
- Click the "Checkout" button.
- Confirm the order form is shown.
- Click the "Place Order" button.
- Confirm that error messages are displayed for the name, address, email, and pay type fields.
- Fill in the Name, Address, and Email text fields.
- Select a Pay type of "Purchase order".
- Click the "Place Order" button.
- Confirm returning to the homepage with a message of "Thank you for your order".

When we collapse all these four tests into a single flow, we end up with a single test that does everything each of the others does, but in a single web transaction.
[pause]
In the GitHub repository, I've left all five tests present, and I leave it as an exercise to the reader to confirm that this holistic test covers all the bases and trust that the other four tests can be deleted.
[slide]

Put it altogether...

```
54  it 'is unified, grandly (with a failure case)' do
55    cs_book = products :coffee
56    ruby_book = products :ruby
57    test_book = products :test
58
59    # Humble beginnings
60    visit '/'
61
62    # First item into cart
63    find("#product_#{cs_book.id}").click
64    find('div#cart').should have_content cs_book.title
65    find('div#cart').should have_content cs_book.price
66
67    # Second item replaces first
68    find("#product_#{ruby_book.id}").click
69    find('div#cart').should have_content ruby_book.title
70    find('div#cart').should have_content ruby_book.price
71
72    # Empty the cart
73    find('div#cart input[value="Empty cart"]').click
74    find('div#cart').should_not have_content ruby_book.title
75    find('div#cart').should_not have_content ruby_book.price
76
77    # Now add something back (can you imagine how tedious this is, manually?!)
78    find("#product_#{test_book.id}").click
79    find('div#cart').should have_content test_book.title
80    find('div#cart').should have_content test_book.price
81
82    # Checking out
83    find('div#cart input[value="Checkout"]').click
84    find('div#main').should have_content 'Please Enter Your Details'
85
86    # Ensure proper errors show up
87    find('input[value="Place Order"]').click
88    find('div#error_explanation').should have_content "Name can't be blank"
89    find('div#error_explanation').should have_content "Address can't be blank"
90    find('div#error_explanation').should have_content "Email can't be blank"
91    find('div#error_explanation').should have_content "Pay type is not included in the list"
92
93    # Fill out the form (demo of scoping behavior)
94    within('form#new_order') do
95      fill_in 'Name', with: 'Daniel Hedrick'
96      fill_in 'Address', with: '1234 Main Street'
97      fill_in 'Email', with: 'daniel@hedrick.org'
98      select 'Check', from: 'Pay type'
99      find('input[value="Place Order"]').click
100   end
101
102   page.should have_content 'Thank you for your order'
103 end
```

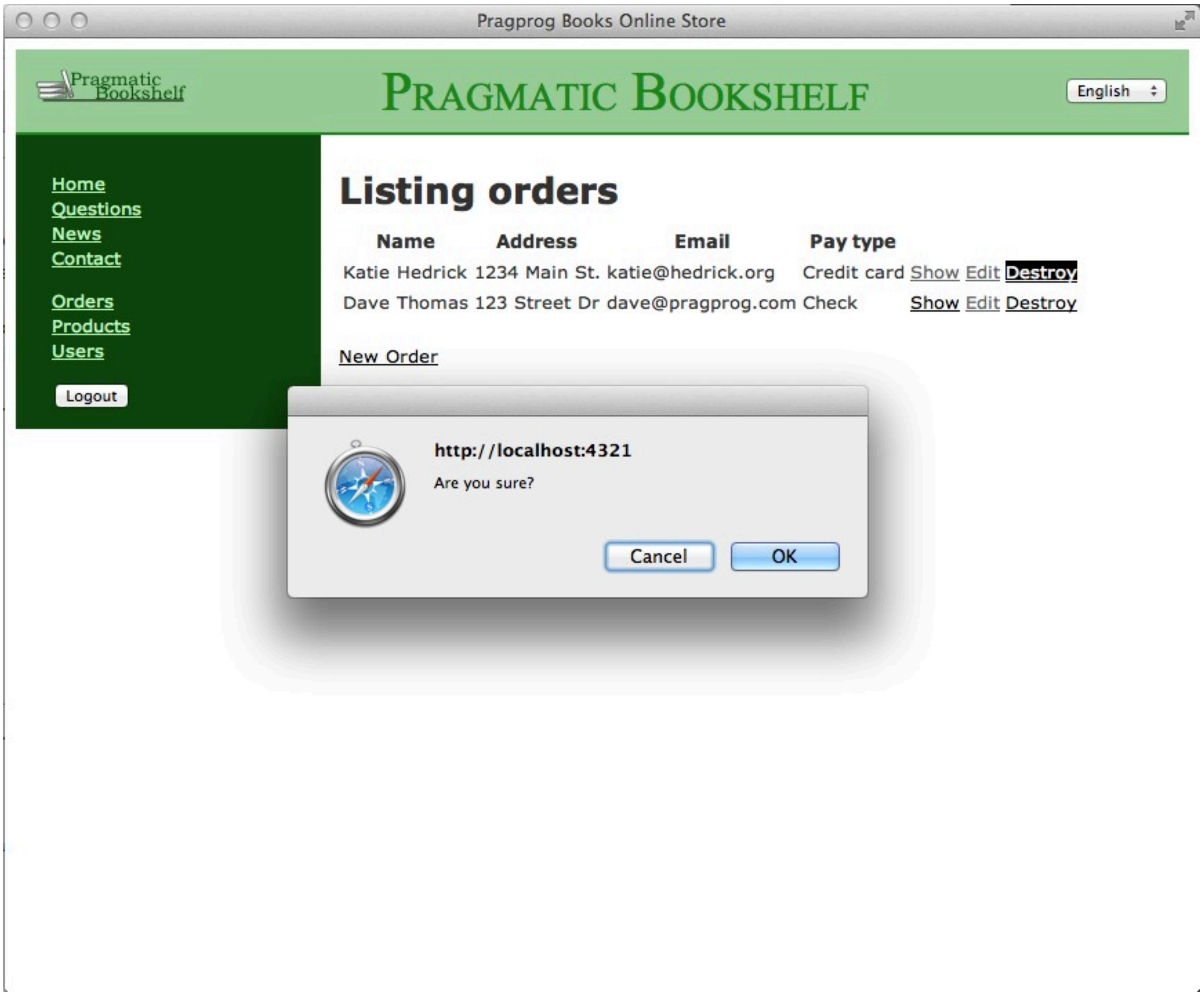
The interesting thing in all of these tests, is their simple use of Capybara's DSL to confirm assertions using a basic HTTP request and response process. The store, as written, uses ajax and javascript, however the code from the book is smart enough not to fail when there's no javascript available.

If you were to actually use the browser to manually perform the "empty the cart" test plan, you'd find that there's a javascript confirmation popup before the cart can actually be deleted. Rather than revise that test, we'll press on to the back office and use Capybara's Selenium driver to test the javascript confirmation.

The Back Office

Selenium is a front-end driver that's built-in to Capybara in its default configuration. Rather than simply accessing the response blob for tests (and DOM inspection), it actually uses Firefox to act as a front-end and drives the tests via honest-to-goodness client-side execution. There are several other Capybara drivers like Selenium that operate "client-side" -- most notably WebKit and Poltergeist.

The primary benefit we get by using Selenium is that it can execute javascript, and thus, will behave exactly like a user's browser.
[slide]



For this application, that means that when we click on a "Destroy" link for a Product, Order, or User, we can expect that the javascript confirmation modal is working the way we intended it.

[slide]

1 Gemfile — View file @ 9da55ca

```
... ...  @@ -14,6 +14,7 @@ end
14  14    group :test, :development do
15  15      gem "rspec-rails", "~> 2.0"
16  16      gem "capybara"
    17  +   gem "database_cleaner"
17  18    end
18  19
19  20    # Gems used only for assets and not required
```

2 spec/spec_helper.rb — View file @ 9da55ca

```
... ...  @@ -24,7 +24,7 @@
24  24      # If you're not using ActiveRecord, or you'd prefer not to run each of your
25  25      # examples within a transaction, remove the following line or assign false
26  26      # instead of true.
27      -   config.use_transactional_fixtures = true
    27  +   #config.use_transactional_fixtures = true
28  28
29  29      # If true, the base class of anonymous controllers will be inferred
30  30      # automatically. This will be the default behavior in future versions of
```

11 spec/support/database_cleaner.rb — View file @ 9da55ca

```
... ...  @@ -0,0 +1,11 @@
    1  +DatabaseCleaner.strategy = :truncation
    2  +
    3  +RSpec.configure do |config|
    4  +  config.use_transactional_fixtures = false
    5  +  config.before :each do
    6  +    DatabaseCleaner.start
    7  +  end
    8  +  config.after :each do
    9  +    DatabaseCleaner.clean
   10  +  end
   11  +end
```

As another bit of house-keeping, please make sure you review this particular commit in the git history, as it was a little bit more involved to get the Selenium driver working. In the Rails environment ‘test’ each action is wrapped in a database transaction, which we need to change. We’ll employ the DatabaseCleaner gem to scrub our tables between each test since we change the behavior of RSpec so that it does NOT use transactional fixtures (transactional in the HTTP request/response sense).

This is a critical change in the test behavior, and if you’re not careful, it’s very easy to spend quite a bit of time struggling with odd test and database behavior only to realize it’s nothing you’ve been doing, but rather the underlying system.

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```ruby
require 'spec_helper'

describe 'backoffice', type: :feature do
  fixtures :products
  fixtures :users

  it 'processes an order', js: true do
    cs_book = products :coffee
    user = users :one

    Order.count.should eq(0)

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find('div#cart input[value="Checkout"]').click
    find('div#main').should have_content 'Please Enter Your Details'
    fill_in 'Name', with: 'Daniel Hedrick'
    fill_in 'Address', with: '1234 Main Street'
    fill_in 'Email', with: 'daniel@hedrick.org'
    select 'Credit card', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    Order.count.should eq(1)

    visit '/login'
```

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```ruby
require 'spec_helper'

describe 'backoffice', type: :feature do
  fixtures :products
  fixtures :users

  it 'processes an order', js: true do
    cs_book = products :cooking
    user = users :one

    Order.count.should eq(0)

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find('div#cart input[value="Checkout"]').click
    find('div#main').should have_content 'Please Enter Your Details'
    fill_in 'Name', with: 'Daniel Hedrick'
    fill_in 'Address', with: '1234 Main Street'
    fill_in 'Email', with: 'daniel@hedrick.org'
    select 'Credit card', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    Order.count.should eq(1)

    visit '/login'
```

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```ruby
require 'spec_helper'

describe 'backoffice', type: :feature do
  fixtures :products
  fixtures :users

  it 'processes an order', js: true do
    cs_book = products :coffee
    user = users :one

    Order.count.should eq(0)

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find('div#cart input[value="Checkout"]').click
    find('div#main').should have_content 'Please Enter Your Details'
    fill_in 'Name', with: 'Daniel Hedrick'
    fill_in 'Address', with: '1234 Main Street'
    fill_in 'Email', with: 'daniel@hedrick.org'
    select 'Credit card', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    Order.count.should eq(1)

    visit '/login'
```

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```ruby
require 'spec_helper'

describe 'backoffice', type: :feature do
  fixtures :products
  fixtures :users

  it 'processes an order', js: true do
    cs_book = products :coffee
    user = users :one

    Order.count.should eq(0)

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find('div#cart input[value="Checkout"]').click
    find('div#main').should have_content 'Please Enter Your Details'
    fill_in 'Name', with: 'Daniel Hedrick'
    fill_in 'Address', with: '1234 Main Street'
    fill_in 'Email', with: 'daniel@hedrick.org'
    select 'Credit card', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    Order.count.should eq(1)

    visit '/login'
```

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```ruby
require 'spec_helper'

describe 'backoffice', type: :feature do
  fixtures :products
  fixtures :users

  it 'processes an order', js: true do
    cs_book = products :coffee
    user = users :one

    Order.count.should eq(0)

    visit '/'
    find("#product_#{cs_book.id}").click
    find('div#cart').should have_content cs_book.title
    find('div#cart input[value="Checkout"]').click
    find('div#main').should have_content 'Please Enter Your Details'
    fill_in 'Name', with: 'Daniel Hedrick'
    fill_in 'Address', with: '1234 Main Street'
    fill_in 'Email', with: 'daniel@hedrick.org'
    select 'Credit card', from: 'Pay type'
    find('input[value="Place Order"]').click
    page.should have_content 'Thank you for your order'

    Order.count.should eq(1)

    visit '/login'
```

Pretty much everything in the back office tests are identical to the store tests, with only a couple notable additions.

The first, we've talked about -- the javascript -- is specified by adding a "js: true" directive to the test.[click]

This will tell Capybara to use the Selenium driver (or whatever javascript-capable driver you've specified) to execute the test. We use it because we want to pretend to interact with the browser's modal js dialog. The other notable behavior we employ in this test is the use of ActiveRecord query methods [click] to confirm counts of records in the database.

The "processes an order" script starts out, just as the store tests, by adding an item to the cart, then processing the checkout and placing the order.

After that comes the meat. [slide]

```
27      visit '/login'
28      page.should have_content 'Please Log In'
29      fill_in 'Name', with: user.name
30      fill_in 'Password', with: 'secret'
31      find('input[value="Login"]').click
32      page.should have_content 'Welcome'
33
34      click_link 'Orders'
35      page.should have_content 'Daniel Hedrick'
36      page.should have_content '1234 Main Street'
37      within('div#main table') do
38        click_link 'Show'
39      end
40      page.should have_content 'Credit card'
41
42      click_link 'Edit'
43      select 'Check', from: 'Pay type'
44      find('input[value="Place Order"]').click
45      page.should have_content 'Order was successfully updated'
46      page.should have_content 'Check'
47
48      click_link 'Back'
49      page.execute_script 'window.confirm = function() { return false; }'
50      click_link 'Destroy'
51
52      click_link 'Orders'
53      page.should have_content 'Daniel Hedrick'
54      page.should have_content '1234 Main Street'
55      page.execute_script 'window.confirm = function() { return true; }'
56      click_link 'Destroy'
57
58      page.should_not have_content 'Daniel Hedrick'
59      page.should_not have_content '1234 Main Street'
60
61      Order.count.should eq(0)
62    end
```

We visit the '/login' route and authenticate using a fixtured user, shown in lines 27-32. Then we confirm that the order placed shows up in the list of orders: lines 34-36. Lines 37-39 shows the use of a scoping method, within, to demonstrate that the behavior can be narrowed down within a DOM block.

Next we edit the order's payment type to confirm the Update function of our CRUD test in lines 42-46.

The last two parts of this test are to cancel our attempt to destroy the record (via a javascript stub returning false on line 49) and then to follow through with the deletion of the record, again by stubbing the javascript with true, this time on line 55.
[slide]

A little bit of javascript

```
48      click_link 'Back'
49      page.execute_script 'window.confirm = function() { return false; }'
50      click_link 'Destroy'
51
52      click_link 'Orders'
53      page.should have_content 'Daniel Hedrick'
54      page.should have_content '1234 Main Street'
55      page.execute_script 'window.confirm = function() { return true; }'
56      click_link 'Destroy'
57
58      page.should_not have_content 'Daniel Hedrick'
59      page.should_not have_content '1234 Main Street'
60
61      Order.count.should eq(0)
```

So, how is it that we go about "stubbing the javascript"? Before we have the test click the "Destroy" link, we shim the client's "window.confirm" function to return false automatically. This simulates the behavior of clicking the 'Cancel' button of the confirmation modal. Once we refresh the page (via 'click_link "Orders"' on line 52), we re-shim the function to return true and click the "Destroy" link again. The last assertions in this test verify that the content is no longer there, and that the Order count is again zero.
[slide]

Users

1. Visit the login screen ("/login")
2. Fill in the Name and Password fields
3. Click the "Login" button
4. Confirm the "Welcome" banner is present
5. Click the "Users" link
6. Click the "New User" link
7. Fill in the Name, Password, and Confirm fields
8. Click the "Create User" button
9. Confirm the flash "User #{name} was successfully created."
10. Click the "Logout" button (left rail)
11. Visit the login screen ("/login")
12. Enter the Name and Password fields
13. Click the "Login" button
14. Confirm the "Welcome" banner is present
15. Click the "Users" link
16. Click the "Destroy" link for the user
17. Click the "OK" button on the popup
18. Confirm the login screen is displayed

With little exception, the CRUD tests for products and users is very similar to the test for managing orders. One unique feature that is worth mentioning here, and that we've documented in our test plan[click], is that when a user is deleted from the database, that user's session is immediately invalidated and no longer has the ability to perform administrative functions.
[slide]

Users

1. Visit the login screen ("/login")
2. Fill in the Name and Password fields
3. Click the "Login" button
4. Confirm the "Welcome" banner is present
5. Click the "Users" link
6. Click the "New User" link
7. Fill in the Name, Password, and Confirm fields
8. Click the "Create User" button
9. Confirm the flash "User #{name} was successfully created."
10. Click the "Logout" button (left rail)
11. Visit the login screen ("/login")
12. Enter the Name and Password fields
13. Click the "Login" button
14. Confirm the "Welcome" banner is present
15. Click the "Users" link
16. Click the "Destroy" link for the user
17. Click the "OK" button on the popup
18. Confirm the login screen is displayed

With little exception, the CRUD tests for products and users is very similar to the test for managing orders. One unique feature that is worth mentioning here, and that we've documented in our test plan[click], is that when a user is deleted from the database, that user's session is immediately invalidated and no longer has the ability to perform administrative functions.
[slide]

```
102   it 'manages users', js: true do
103     user = users :one
104
105     visit '/login'
106     page.should have_content 'Please Log In'
107     fill_in 'Name', with: user.name
108     fill_in 'Password', with: 'secret'
109     find('input[value="Login"]').click
110     page.should have_content 'Welcome'
111
112     click_link 'Users'
113     click_link 'New User'
114     fill_in 'Name', with: 'newuser'
115     fill_in 'Password', with: 'apass'
116     fill_in 'Confirm', with: 'apass'
117     find('input[value="Create User"]').click
118     page.should have_content 'User newuser was successfully created.'
119
120     find('input[value="Logout"]').click
121
122     visit '/login'
123     page.should have_content 'Please Log In'
124     fill_in 'Name', with: 'newuser'
125     fill_in 'Password', with: 'apass'
126     find('input[value="Login"]').click
127     page.should have_content 'Welcome'
128
129     click_link 'Users'
130
131     page.execute_script 'window.confirm = function() { return true; }'
132     within('div#main table tr:last') do
133       click_link 'Destroy'
134     end
135
136     page.should have_content 'Please Log In'
137   end
```

In our "manages users" test, we log in using our fixture'd user, then we create a new user; log out; then log back in using this newly created account. The last step of this test procedure is to delete the user (yes, we're effectively deleting ourself from the database).
[click]
Once that happens, the user should immediately be redirected to the login screen.

```
102   it 'manages users', js: true do
103     user = users :one
104
105     visit '/login'
106     page.should have_content 'Please Log In'
107     fill_in 'Name', with: user.name
108     fill_in 'Password', with: 'secret'
109     find('input[value="Login"]').click
110     page.should have_content 'Welcome'
111
112     click_link 'Users'
113     click_link 'New User'
114     fill_in 'Name', with: 'newuser'
115     fill_in 'Password', with: 'apass'
116     fill_in 'Confirm', with: 'apass'
117     find('input[value="Create User"]').click
118     page.should have_content 'User newuser was successfully created.'
119
120     find('input[value="Logout"]').click
121
122     visit '/login'
123     page.should have_content 'Please Log In'
124     fill_in 'Name', with: 'newuser'
125     fill_in 'Password', with: 'apass'
126     find('input[value="Login"]').click
127     page.should have_content 'Welcome'
128
129     click_link 'Users'
130
131     page.execute_script 'window.confirm = function() { return true; }'
132     within('div#main table tr:last') do
133       click_link 'Destroy'
134     end
135
136     page.should have_content 'Please Log In'
137   end
```

In our "manages users" test, we log in using our fixture'd user, then we create a new user; log out; then log back in using this newly created account. The last step of this test procedure is to delete the user (yes, we're effectively deleting ourself from the database).
[click]
Once that happens, the user should immediately be redirected to the login screen.

Debugging

```
click_link('foo')
print page.html
page.should have_content('bar')

save_and_open_page

save_screenshot(‘myfile.png’)
```

Capybara has several features for debugging tests (and applications). The feature I found most useful was the ability to grab the entire document response in a single string and print it out. In this example, if the assertion for the content ‘bar’ fails, the print statement outputs to standard out and you can review the document to see what might be the issue.

click_link('foo')
print page.html
page.should have_content('bar')

Additionally, the function #save_and_open_page is supposed to take a snapshot of the page as it currently exists and open it for inspection. I tried once to use this and was unsuccessful. The page.html was always sufficient and so I did not explore it further. Another useful debugging feature is that, with drivers that support it (such as WebKit or Selenium), you can call page.save_screenshot('myfile.png') to save a screenshot.
[slide]

AJAX

We saw some basic testing against ajax functionality in the back-office tests but I wanted to take just a second to drill into that... If you've ever used a testing tool like WinRunner, you might have been wondering how Capybara deals with the intrinsic pauses during network activity in the ajax calls. Wonderfully, Capybara automatically deals with dynamic pages by waiting for elements to appear on the page. The default wait is 2 seconds, but can be adjusted in Capybara's configuration. And this is very powerful. If, for example, you have the following: [click]
find('#foo').click_link('bar')

Capybara's engine will monitor the #foo for the link 'bar'. #foo could leave the page (via ajax/DOM manipulation) and then return, still without the 'bar' link, and then inject it into the DOM. As long as you're within your configured default wait time, Capybara will click that link when it arrives.

In the tests that I wrote for Depot application, I never had any sort of issue with DOM manipulation, and I expect that with the debugging tools and a focus on simplicity in your test syntax, it should be pretty capable. Ultimately, if it's hard to write tests for something, that's a code smell in and of itself.

AJAX

```
find(‘#foo’).click_link(‘bar’)
```

We saw some basic testing against ajax functionality in the back-office tests but I wanted to take just a second to drill into that... If you’ve ever used a testing tool like WinRunner, you might have been wondering how Capybara deals with the intrinsic pauses during network activity in the ajax calls. Wonderfully, Capybara automatically deals with dynamic pages by waiting for elements to appear on the page. The default wait is 2 seconds, but can be adjusted in Capybara's configuration. And this is very powerful. If, for example, you have the following: [click]
find('#foo').click_link('bar')

Capybara's engine will monitor the #foo for the link 'bar'. #foo could leave the page (via ajax/DOM manipulation) and then return, still without the ‘bar’ link, and then inject it into the DOM. As long as you're within your configured default wait time, Capybara will click that link when it arrives.

In the tests that I wrote for Depot application, I never had any sort of issue with DOM manipulation, and I expect that with the debugging tools and a focus on simplicity in your test syntax, it should be pretty capable. Ultimately, if it's hard to write tests for something, that's a code smell in and of itself.

Core Behavior of DSL:

- visit, find, click
- fill_in, choose, check, select
- page.should have_content('foo')

So, Capybara in a not-so-small nutshell. The core of its behavior lies in the DSL's "visit", "find", and "click" functions. To populate forms, use the "fill_in", "choose", "check", and "select" functions. To assert that the application responds with what we expect, we use RSpec's "should" syntax to verify that certain content exists. To learn more about the form-handling features in the DSL for "choose"ing radio buttons, "check"ing check-boxes, "select"ing dropdowns, and attaching files, review Capybara's README.md on GitHub.
[slide]

Additionally, when switching to using a javascript driver, make sure you consider how you’re doing database transactions and that they coincide properly with your web transactions (as in, you probably want data to hang around longer than a single HTTP request).
[slide]

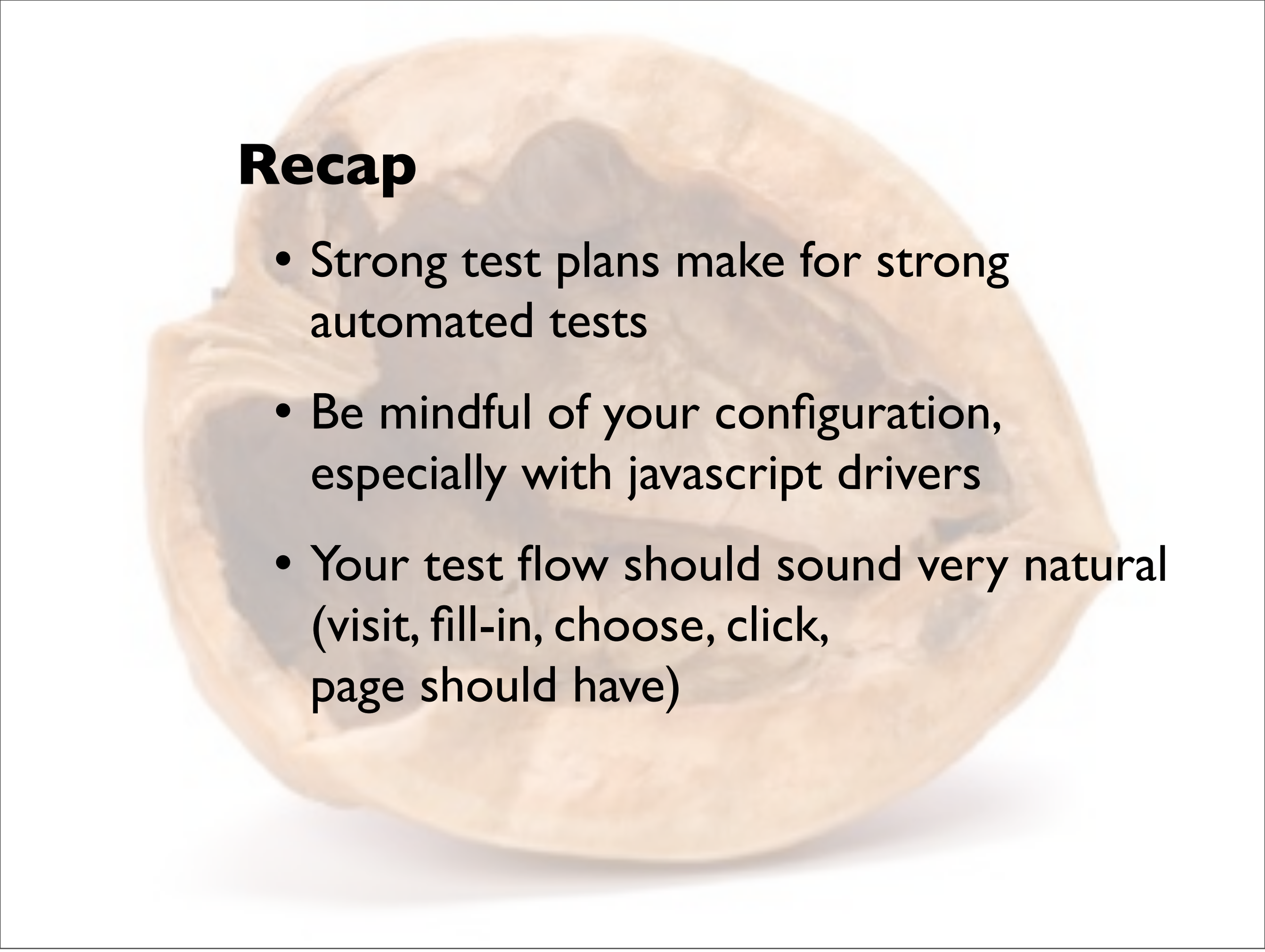
Have a Test Plan

- documents user behavior
- provides a “script” that can be automated
- Start with a “smoke test” to verify basic application behaviors.

And going back to the beginning of the discussion, be very thoughtful about how you intend to construct your integration tests. The easiest way to do that is with an on-paper test plan.
The “smoke tests” I built here not only help me better understand how my application works, but also gives me a safety net I can count on to quickly confirm the application is behaving correctly.

That's pretty much it. Simple setup, simple syntax, simple automation. And now, after every build, we know whether our Depot is "basically working" within just a minute rather than the 10 minutes it used to take us to run through our smoke test.
[slide]
Realistically, we spent only a fraction of this discussion talking about Capybara's feature set, but all-in-all, I think it's comprehensive. So remember:

* Strong test plans make for strong automated tests
* Be mindful of your configuration, especially with javascript drivers
* Your test flow should sound very natural (visit, fill-in, choose, click, page.should have).



Realistically, we spent only a fraction of this discussion talking about Capybara's feature set, but all-in-all, I think it's comprehensive. So remember:

* Strong test plans make for strong automated tests
* Be mindful of your configuration, especially with javascript drivers
* Your test flow should sound very natural (visit, fill-in, choose, click, page.should have).

[slide]

Daniel Hedrick, @dcoder
daniel@hedrick.org

https://github.com/dcoder2099
/intro_to_capybara

Thank you for your time this evening. The code, including annotated commits of all the tests I wrote, is available on my GitHub page. If you found this useful, please tell others; if you think I bombed, tell me.

Any questions?