



C o m m u n i t y E x p e r i e n c e D i s t i l l e d

Application Testing with Capybara

Confidently implement automated tests for web applications
using Capybara

Matthew Robbins

www.allitebooks.com

[PACKT] open source*
PUBLISHING
community experience distilled

Teste de aplicativo com capivara

Implemente com confiança testes automatizados para
aplicativos da web usando Capivara

Mateus Robbins



BIRMINGHAM - MUMBAI

Teste de aplicativo com Capivara

Copyright © 2013 Packt Publishing

Todos os direitos reservados. Nenhuma parte deste livro pode ser reproduzida, armazenada em um sistema de recuperação ou transmitida de qualquer forma ou por qualquer meio, sem a permissão prévia por escrito do editor, exceto no caso de breves citações incorporadas em artigos críticos ou resenhas.

Todo esforço foi feito na preparação deste livro para garantir a precisão das informações apresentadas. No entanto, as informações contidas neste livro são vendidas sem garantia, expressa ou implícita. Nem o autor, nem a Packt Publishing e seus revendedores e distribuidores serão responsabilizados por quaisquer danos causados ou supostamente causados direta ou indiretamente por este livro.

A Packt Publishing se esforçou para fornecer informações de marca registrada sobre todas as empresas e produtos mencionados neste livro pelo uso apropriado de maiúsculas. No entanto, a Packt Publishing não pode garantir a precisão dessas informações.

Primeira publicação: setembro de 2013

Referência de produção: 1160913

Publicado por Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, Reino Unido.

ISBN 978-1-78328-125-1

www.packtpub.com

Imagen da capa por VigilancePrime ([en.wikipedia](https://en.wikipedia.org))

Créditos

Autor

Mateus Robbins

Coordenador de projeto

Sherin Padayatty

Revisores

Yavor Atanasov

Revisores

Clyde Jenkins

Graham Lyons

Christopher Smith

Daniel Morrison

Indexador

Priya Subramani

Editor de aquisição

Aarthi Kumaraswamy

Gráficos

Ronak Dhruv

Editor comissionado

Poonam Jain

Coordenador de produção

Arvindkumar Gupta

Editores técnicos

Hardik B. Soni

trabalho de capa

Krutika Parab

Arvindkumar Gupta

Editores de cópia

Adithi Shetty

Sayanee Mukherjee

Alfida Paiva

Sobre o autor

Matthew Robbins é um desenvolvedor experiente em testes, tendo passado muitos anos lutando com ferramentas de automação de testes disponíveis comercialmente. Ele passou os últimos cinco anos imerso no desenvolvimento de estruturas robustas de automação de teste usando ferramentas de código aberto. Ele trabalhou extensivamente com a BBC desenvolvendo estruturas e ferramentas de automação de teste em sua plataforma da web e continua a trabalhar na indústria de mídia para outras emissoras de alto nível. Além da automação de teste, ele é apaixonado por se tornar mais produtivo no Vim e aprender sobre os componentes internos do navegador da web. Ele também bloga regularmente em <http://opensourcetester.co.uk>.

Gostaria de agradecer a Catherine, Jared e Leon, minha maravilhosa família, por todo o apoio. Também agradeço imensamente à equipe do BBC Frameworks, especialmente Pete, Graham e Yavor, por me iniciarem nesta jornada.

Sobre os Revisores

Yavor Atanasov é um engenheiro de software com experiência profunda em todo o espectro do desenvolvimento da Web em grande escala. Ele é da Bulgária, atualmente morando e trabalhando em Londres. Ele viu as peculiaridades do desenvolvimento de JavaScript do lado do cliente e a importância de arquitetar e escrever sistemas de backend eficientes. Práticas ágeis, abordagem orientada a testes e comportamento para o desenvolvimento de software são parte fundamental de seu trabalho. Ele também experimentou a complexidade do teste de aceitação de sistemas multicamadas consideráveis.

Graham Lyons é um engenheiro de software que trabalha na Web há cerca de seis anos. Atualmente trabalhando na plataforma da BBC, ele gosta de códigos elegantes e bem testados, escritos em várias linguagens, e provavelmente passa muito tempo pensando em soluções para problemas de engenharia. Quando não está fazendo isso, gosta de ar fresco e de um bom café e atualmente está planejando seu casamento com uma senhora muito paciente.

Daniel Morrison é o fundador da Collective Idea (<http://collectiveidea.com>), uma consultoria de desenvolvimento de software em Holland, Michigan. Na Collective Idea, Daniel trabalhou com empresas da Fortune 50 e desenvolveu software para fabricantes de automóveis, startups do Vale do Silício e tudo mais. Ele escreve muitos aplicativos da Web e ministra cursos de desenvolvimento de software em todo o mundo.

www.PacktPub.com

Arquivos de suporte, eBooks, ofertas de desconto e muito mais Você pode visitar www.PacktPub.com para arquivos de suporte e downloads relacionados ao seu livro.

Você sabia que a Packt oferece versões eBook de todos os livros publicados, com arquivos PDF e ePub disponíveis? Você pode atualizar para a versão eBook em www.PacktPub.com e, como cliente do livro impresso, tem direito a um desconto na cópia do eBook. Entre em contato conosco em service@packtpub.com para mais detalhes.

Em www.PacktPub.com, você também pode ler uma coleção de artigos técnicos gratuitos, inscrever-se em uma variedade de boletins informativos gratuitos e receber descontos e ofertas exclusivas nos livros e e-books da Packt.



<http://PacktLib.PacktPub.com>

Você precisa de soluções instantâneas para suas questões de TI? PacktLib é a biblioteca de livros digitais online da Packt. Aqui, você pode acessar, ler e pesquisar toda a biblioteca de livros da Packt.

Por que se inscrever? •

Totalmente pesquisável em todos os livros publicados pela Packt •

Copie e cole, imprima e marque o conteúdo • Sob demanda e acessível via navegador da web

Acesso gratuito para titulares de contas Packt

Se você tiver uma conta com Packt em www.PacktPub.com, você pode usar isso para acessar PacktLib hoje e ver nove livros totalmente gratuitos. Basta usar suas credenciais de login para acesso imediato.

Índice

Prefácio	1
<hr/>	
Capítulo 1: Seu primeiro cenário com capivara	5
Instalando Capivara	5
Preparando seu sistema	6
Instalando gemas com RubyGems	6
Instalando gemas com o Bundler	7
Instalando as bibliotecas do sistema	8
Instalando Capivara	8
Usando RubyGems	8
Usando o empacotador	10
Instalando Pepino e Selênio	11
Pepino-Rails	13
Seu primeiro cenário – uma pesquisa no YouTube	13
Resumo	18
<hr/>	
Capítulo 2: Dominando a API	19
Localizando elementos com XPath e CSS	19
Seletor padrão na Capivara	20
Uma ajuda com os seletores	21
Navegação	22
Clicar em links ou botões	22
Envio de formulários	24
Caixas de seleção e botões de opção	25
Localizadores, escopo e várias correspondências	28
Múltiplas correspondências	30
Estratégias de correspondência	30
Visibilidade do elemento	33
Escopo	34

Índice

Afirmando e consultando	35
Matchers e RSpec	35
Refinando localizadores e compensadores	38
Verificando os valores dos atributos	40
Resumo	40
Capítulo 3: Testando aplicativos Rails e Sinatra	41
Entendendo a interface do Rack	41
Capivara e Rack::Teste	43
Testando um aplicativo Sinatra Arquivo de aplicativo Sinatra – app.rb Modelo de formulário – form.erb Modelo de resultados – result.erb	45 45 45 47
Testando com Rack::Test	48
Qual driver usar e quando?	51
Uma nota sobre Rails/RSpec e Resumo do Capivara	52
Capítulo 4: Lidando com Ajax, JavaScript e Flash	55
Ajax e JavaScript assíncrono	55
Capivara e JavaScript assíncrono	56
Métodos que lidam com JavaScript assíncrono	59
localizadores	59
Matchers	59
Pegadinhas	60
Flash e HTML5 – elementos de caixa preta	62
Flash	63
Expondo uma API testável	63
Páginas de teste – eis o poder!	65
Componentes de teste "in situ"	68
Resumo	71
Capítulo 5: Tópicos Ninja	73
Usando Capivara fora do Pepino	73
Incluindo os módulos	74
Usando a sessão diretamente	75
Capivara e estruturas de teste populares	76
Cucumber	76
RSpec	77
Test::Unit	77
MiniTest::Spec	77
Interações avançadas e acesso direto ao driver	78
Usando o método nativo	78
Acessando métodos do driver usando browser.manage	79
	80

Índice

Configuração avançada do driver	81
O ecossistema do motorista	82
Capivara-WebKit	83
poltergeist	83
Capivara- Mecanizar	84
Capivara-Celeridade	84
Resumo	85
Índice	87

Machine Translated by Google

Prefácio

Um dos meus colegas uma vez descreveu a comunidade Ruby como "Test Infected" e se alguma biblioteca simboliza isso é a Capybara, que ganhou popularidade exponencialmente desde que foi lançada pela primeira vez. A comunidade Ruby certamente deve ao seu criador *Jonas Nicklas* muitos agradecimentos por trazer paz e harmonia a muitas bases de código de automação de teste em todo o mundo.

A prova do sucesso do Capybara é a maneira como seu uso se espalhou muito além de apenas testar aplicativos Rails e agora suporta o teste de muitos aplicativos da Web escritos em uma ampla variedade de linguagens e estruturas. A funcionalidade do Capivara também foi replicada em outras linguagens além do Ruby, novamente destacando o quão poderoso é o conceito.

Então, o que é Capivara?

O Capybara fornece uma linguagem específica de domínio para automação de teste; esse DSL estende o estilo BDD legível por humanos de estruturas como Cucumber e RSpec no próprio código de automação. Por exemplo, abrir um navegador e navegar até um URL é tão simples quanto visitar <http://google.com>. Esta é uma grande melhoria sobre o típico APIs de teste.

Além disso, o Capivara nos permite escrever testes uma vez e executá-los em qualquer driver compatível. O ecossistema do driver é vibrante e a troca de bibliotecas é tão simples quanto adicionar uma joia adicional e fazer uma alteração de uma linha em seu código.

Finalmente, você pode acabar com a escrita de métodos sob medida que esperam que o conteúdo se torne visível ou adicionar instruções de suspensão aos seus testes; Capivara lida com JavaScript assíncrono sem que o usuário perceba.

Capivara é literalmente o seu balcão único para automação de testes.

Prefácio

O que este livro cobre Capítulo 1, *Seu primeiro cenário com Capivara*, cobre a instalação e configuração de seu primeiro cenário usando Capivara.

O Capítulo 2, *Dominando a API*, fornece um mergulho profundo na API do Capivara para interagir com páginas da web.

O Capítulo 3, *Testando aplicativos Rails e Sinatra*, nos ajuda a explorar como o Capivara é particularmente adequado para testar aplicativos implementados usando Rails ou Sinatra.

O Capítulo 4, *Lidando com Ajax, JavaScript e Flash*, aborda como lidar com JavaScript assíncrono e como usar o Capivara para testar componentes de caixa preta, como Flash ou HTML5 Canvas, Áudio e Vídeo.

O Capítulo 5, *Tópicos Ninja*, nos ajuda a usar o Capybara fora do Cucumber em estruturas sob medida, dentro de estruturas de teste populares como RSpec e explora algumas alternativas para os drivers integrados do Capybara.

O que você precisa para este livro

Este livro e os exemplos foram desenvolvidos usando Ruby-1.9.3p237, RubyGems 1.8.23 e, mais importante, Capybara 2.1.0, que introduziu algumas mudanças significativas. Todas as outras dependências serão baixadas pelo RubyGems ou pelo Bundler quando você instalar o Capybara. Também usaremos Cucumber e RSpec, cujas versões mais recentes devem ser compatíveis com Capivara 2.1.0 e superior.

A quem se destina este livro

Este livro é

para desenvolvedores e testadores que, com alguma exposição ao Ruby, desejam saber como testar seus aplicativos usando o Capybara e seus drivers compatíveis, como Selenium WebDriver e Rack::Test. Os exemplos são deliberadamente mantidos simples e a marcação HTML de exemplo é sempre incluída para que os leitores possam copiar os exemplos para praticar e experimentar em sua própria máquina.

Convenções Neste

livro, você encontrará vários estilos de texto que distinguem diferentes tipos de informação. Aqui estão alguns exemplos desses estilos e uma explicação de seu significado.

Prefácio

Palavras de código em texto, nomes de tabelas de banco de dados, nomes de pastas, nomes de arquivo, extensões de arquivo, nomes de caminho, URLs fictícios, entrada do usuário e identificadores do Twitter são mostrados a seguir: "A única fonte de confusão aqui pode ser o uso da string literal search_query em o método fill_in ."

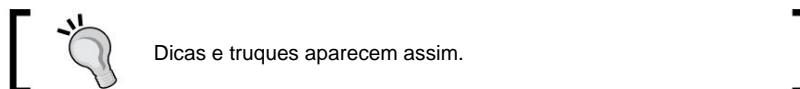
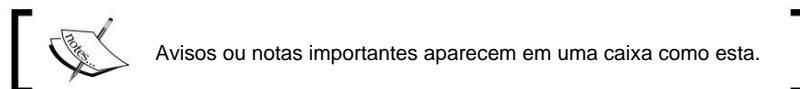
Um bloco de código é definido da seguinte forma:

```
<div id="principal">
  <div class="seção">
    <a id="myanchor" title="myanchortitle" href="#">Clique aqui
      Âncora</a> </
  div> </div>
```

Qualquer entrada ou saída de linha de comando é escrita da seguinte forma:

```
$ rubi -v
ruby 1.9.3p327 (2012-11-10 revisão 37606) [x86_64-darwin11.4.2]
```

Novos termos e palavras importantes são mostrados em negrito. Palavras que você vê na tela, em menus ou caixas de diálogo, por exemplo, aparecem no texto assim: "Depois disso, precisamos inserir nossos termos de pesquisa e clicar no botão **Pesquisar**".



feedback do leitor

O feedback dos nossos leitores é sempre bem-vindo. Deixe-nos saber o que você pensa sobre este livro - o que você gostou ou pode não ter gostado. O feedback dos leitores é importante para desenvolvermos títulos que realmente aproveitem ao máximo.

Para nos enviar um feedback geral, basta enviar um e-mail para feedback@packtpub.com e mencionar o título do livro no assunto de sua mensagem.

Se houver um tópico no qual você seja especialista e estiver interessado em escrever ou contribuir para um livro, consulte nosso guia do autor em www.packtpub.com/authors.

Prefácio

Suporte ao cliente

Agora que você é o orgulhoso proprietário de um livro Packt, temos várias coisas para ajudá-lo a obter o máximo de sua compra.

Baixando o código de exemplo

Você pode baixar os arquivos de código de exemplo para todos os livros Packt que comprou em sua conta em <http://www.packtpub.com>. Se você comprou este livro em outro lugar, visite <http://www.packtpub.com/support> e registre-se para receber os arquivos por e-mail diretamente para você.

Errata Embora

tenhamos tomado todos os cuidados para garantir a precisão de nosso conteúdo, erros acontecem. Se você encontrar um erro em um de nossos livros - talvez um erro no texto ou no código - ficaríamos gratos se você nos relatasse isso. Ao fazer isso, você pode salvar outros leitores da frustração e nos ajudar a melhorar as versões subsequentes deste livro.

Se você encontrar alguma errata, informe-a visitando <http://www.packtpub.com/submit-errata>, selecionando seu livro, clicando no link **do formulário de envio de errata** e inserindo os detalhes de sua errata. Depois que suas erratas forem verificadas, seu envio será aceito e as erratas serão carregadas em nosso site ou adicionadas a qualquer lista de errata existente, na seção Errata desse título. Qualquer errata existente pode ser visualizada selecionando seu título em <http://www.packtpub.com/support>.

Pirataria

A pirataria de material protegido por direitos autorais na Internet é um problema constante em todas as mídias. Na Packt, levamos muito a sério a proteção de nossos direitos autorais e licenças. Se você encontrar cópias ilegais de nossos trabalhos, de qualquer forma, na Internet, forneça-nos o endereço do local ou o nome do site imediatamente para que possamos buscar uma solução.

Entre em contato conosco em copyright@packtpub.com com um link para o material pirata suspeito.

Agradecemos sua ajuda na proteção de nossos autores e nossa capacidade de fornecer conteúdo valioso.

Dúvidas Você pode

entrar em contato conosco pelo e-mail question@packtpub.com se tiver algum problema com qualquer aspecto do livro, e faremos o possível para resolvê-lo.

1

Seu primeiro cenário com capivara

O Capybara traz dois ingredientes principais para a automação de testes: código legível por humanos por meio de uma elegante **linguagem específica de domínio (DSL)** e a capacidade de escrever uma vez e executar em vários drivers, como **Selenium WebDriver** ou aplicativos **Rack::Test** for Rails/ Sinatra. Ao longo deste livro veremos como a Capivara pode aumentar muito a resiliência de nossas provas e potencializar nossa produtividade.

Neste capítulo, iremos percorrer a instalação do Capybara e começar a trabalhar imediatamente com um cenário simples.

Especificamente, abordaremos o seguinte:

- Garantir que Ruby, RubyGems e Bundler estejam disponíveis • Garantir que você tenha as bibliotecas de sistema necessárias disponíveis • Instalar a gem Capivara • Implementar seu primeiro cenário com Capivara, Pepino e

Selenium WebDriverName

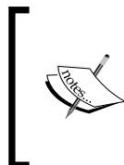
Instalando Capivara

Instalar o Capybara não é diferente de instalar qualquer outra gem Ruby; se você já fez algum desenvolvimento Ruby no passado, é provável que você tenha todos os pré-requisitos, e esse processo será direto.

Seu primeiro cenário com capivara

Preparando seu sistema

Capivara é uma gem Ruby e, como tal, precisamos garantir que Ruby esteja disponível no sistema.



Quando você está aprendendo uma nova biblioteca, faz sentido usar o Ruby padrão (ao contrário do Ruby que roda em uma plataforma diferente, como JRuby ou Iron Ruby) e executar tudo a partir da linha de comando. Isso simplesmente limita a quantidade de coisas que podem dar errado devido às peculiaridades de uma plataforma com menos suporte ou de um IDE excessivamente complicado.

Abra um prompt de comando e verifique a versão do seu interpretador Ruby:

```
$ rubi -v  
ruby 1.9.3p327 (2012-11-10 revisão 37606) [x86_64-darwin11.4.2]
```

Se você vir algo maior que 1.9, perfeito! Isso significa que a versão correta do Ruby está instalada e disponível para você usar.

Se você vir o comando não encontrado, isso significa que o Ruby não está instalado ou o sistema não pode encontrá-lo. Se você acha que instalou o Ruby, tente modificar sua variável PATH e adicione o local do executável do Ruby.

Se você vir algo menor que 1.9, atualize sua versão atual do Ruby para 1.9 ou superior. Capivara tem muitas dependências e não suporta mais versões do Ruby anteriores a 1.9.

Instalando gemas com RubyGems

Como a maioria das linguagens Ruby tem seu próprio mecanismo para gerenciar bibliotecas de código. **RubyGems** é o software usado para gerenciar as gems que são bibliotecas e o Capybara é uma gem Ruby em si.

O aplicativo RubyGems normalmente é instalado quando você instala o Ruby pela primeira vez, mas vale a pena verificar novamente se você o possui.

No prompt de comando, verifique a versão do RubyGems instalada executando o seguinte comando:

```
$ gem -v  
1.8.23
```

Se você vir algo maior que a versão 1.5.0, está pronto.

Se você vir a mensagem de comando não encontrado , você precisa instalar o RubyGems ou adicionar o executável à sua variável PATH .

Se você vir algo menor que a versão 1.5.0, atualize a versão executando o seguinte comando:

atualização de gem —sistema

Instalando gemas com o Bundler

Embora você possa instalar e usar o Capybara tranquilamente sem usar o **Bundler**, vale a pena cobrir isso porque o Bundler está se tornando onipresente no ecossistema Ruby e é muito provável que você queira usá-lo para gerenciar as dependências do seu projeto.

O próprio Bundler é uma gema Ruby e aplica uma camada de gerenciamento de dependência mais refinado sobre RubyGems. Com RubyGems você só pode ter uma versão de uma gem instalada em seu sistema; com Bundler você pode isolar dependências para um projeto específico.

Execute o seguinte comando para instalar o Bundler:

empacotador de instalação de gem

No diretório do seu projeto, crie um arquivo chamado Gemfile com o seguinte conteúdo:

```
fonte 'https://rubygems.org'
```

```
gema 'capivara'
```

Em seguida, em um prompt de comando dentro desse diretório, execute:

instalação do pacote

Isto irá instalar e vincular as gems que você especificou em seu Gemfile (bem como todas as suas dependências) com seu projeto atual. Isso evitaria que você interrompa accidentalmente outro projeto que possa, por exemplo, depender de uma versão anterior da gema **Nokogiri** e geralmente facilitará muito sua vida.

O Bundler permite que você declare requisitos de versão específicos em seu projeto Gemfile e fornece muitas outras opções, como recuperar uma gem diretamente de um repositório GitHub. Você também pode agrupar gems em um diretório local em seu projeto. Consulte <http://gembundler.com> para obter mais detalhes sobre esta joia incrível.

Seu primeiro cenário com capivara

Instalando as bibliotecas do sistema

Em algumas plataformas, certas gems dependem de bibliotecas do sistema. Isso geralmente é feito por motivos de desempenho. Ruby é uma linguagem interpretada, portanto, tarefas como análise de XML podem ser lentas; portanto, faz sentido delegar essa tarefa a uma biblioteca do sistema.

No Windows, você não precisará se preocupar com isso, embora tenha que garantir que possui o Ruby DevKit instalado; consulte <http://rubyinstaller.org/add-ons/devkit> para obter instruções detalhadas sobre como fazer isso.

O Capybara depende do Nokogiri, o popular analisador de XML baseado em Ruby. Isso, por sua vez, precisa que as seguintes bibliotecas do sistema estejam disponíveis:

- libxml2
- libxml2-dev
- libxslt
- libxslt-dev



A versão mais recente do Nokogiri agora inclui essas dependências dentro da própria gem. No entanto, ainda vale a pena instalar as bibliotecas do sistema globalmente, pois você certamente encontrará projetos que dependem de versões do Nokogiri anteriores a 1.6.0.



A forma como você os instala em um sistema específico será diferente, por exemplo, apt-get para Ubuntu, yum para Red Hat ou brew para Mac OS X.

Instalando Capivara

Seu sistema agora está pronto para uma instalação indolor do Capivara. A maneira como você instala a gem dependerá se você optar por usar RubyGems diretamente ou Bundler em seu projeto.

Usando RubyGems

Se você decidir ficar sem o Bundler, instalar o Capivara é tão simples quanto:

gem install capivara

Capítulo 1

Se tudo correr bem, você deverá ver uma saída como a seguinte em seu prompt de comando. A saída precisa pode diferir ligeiramente dependendo de quantas dependências você já instalou:

Buscando: mime-types-1.25.gem (100%)

Obtendo: rack-1.5.2.gem (100%)

Obtendo: rack-test-0.6.2.gem (100%)

Obtendo: xpath-2.0.0.gem (100%)

Buscando: capivara-2.1.0.gem (100%)

IMPORTANTE! Alguns dos padrões foram alterados no Capivara 2.1. Se você está enfrentando falhas,

por favor, reverta para o comportamento antigo definindo:

```
Capivara.configure do |config|
  config.match = :one
  config.exact_options = verdadeiro
  config.ignore_hidden_elements = verdadeiro
  config.visible_text_only = verdadeiro
fim
```

Se você estiver migrando do Capivara 1.x, tente:

```
Capivara.configure do |config|
  config.match = :prefer_exact
  config.ignore_hidden_elements = false
fim
```

Detalhes aqui: <http://www.elabs.se/blog/60-introducing-capybara-2-1>

mini_portile-0.5.1 instalado com sucesso

Nokogiri-1.6.0 instalado com sucesso

Seu primeiro cenário com capivara

Mime-types-1.25 instalado com sucesso

Rack-1.5.2 instalado com sucesso

rack-test-0.6.2 instalado com sucesso

xpath-2.0.0 instalado com sucesso

Capivara-2.1.0 instalado com sucesso

7 gemas instaladas

Usando o empacotador

Se você estiver usando o Bundler, certifique-se de ter um arquivo chamado Gemfile no diretório raiz do diretório do seu projeto e que contenha o seguinte:

fonte '<https://rubygems.org>'

gema 'capivara'

Em seguida, instale as gems necessárias executando o seguinte comando:

instalação do pacote

Se tudo for bem-sucedido, você deverá ver uma saída como a seguinte:

\$ instalação do pacote

Obtendo metadados gem de <https://rubygems.org/>.....

Instalando tipos MIME (1.25)

Instalando o mini_portile (0.5.1)

Instalando nokogiri (1.6.0)

Instalação do rack (1.5.2)

Instalação de teste de rack (0.6.2)

Instalando xpath (2.0.0)

Instalando a capivara (2.1.0)

Usando empacotador (1.3.5)

Seu pacote está completo!

Instalando Pepino e Selênio

Capivara é simplesmente uma API que fornece uma camada de abstração sobre sua biblioteca de automação real. Se ajudar, pense em Capivara como seu tradutor; você diz para fazer algo e traduz um comando elegante e agradável na API do seu driver fornecido (que pode ser muito menos amigável).

Portanto, para usar esse tradutor, precisamos ter uma maneira de dizer o que fazer e também uma API de biblioteca de automação para traduzir.

A Capivara é uma biblioteca muito flexível e, ao longo deste livro, a veremos sendo usada em diversos ambientes; no entanto, de longe, o caso de uso mais comum é empregar o Cucumber como executor de teste com o Capybara dirigindo o Selenium WebDriver para realizar a automação do navegador.

O Cucumber permite a execução de cenários de desenvolvimento orientado a comportamento (BDD) escritos na sintaxe Gherkin para conduzir seus testes. Se você não está muito familiarizado com o Cucumber, <http://cukes.info> deve ser seu primeiro porto de escala, mas não se preocupe, é muito direto e iremos criar seu primeiro cenário.

Aqui está um exemplo de cenário Cucumber, que vamos automatizar:

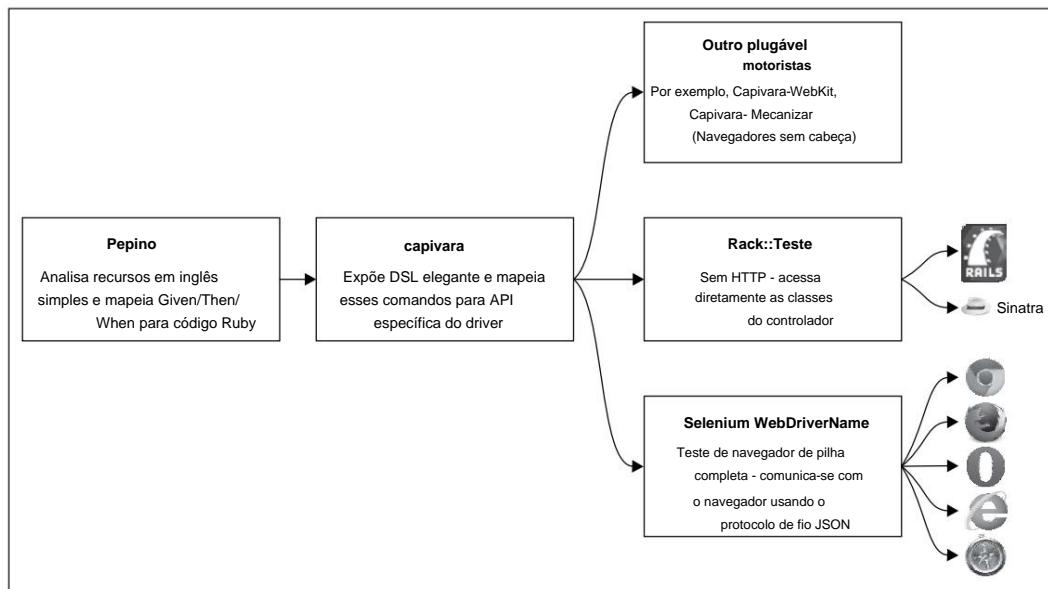
Recurso: Pesquisar vídeos no YouTube

```
Cenário: Pesquisar vídeos de grandes roedores
  Dado que estou na página inicial do YouTube
  Quando procuro por "capivara"
  Em seguida, vídeos de grandes roedores são retornados
```

Quando o Cucumber é invocado, ele analisa o cenário inglês simples e, usando expressões regulares, corresponde cada linha a uma linha real de código Ruby, chamada de **definição de etapa**. Usaremos o Capivara para implementar essas etapas. O Capivara então fará a comunicação com o Selenium WebDriver, que abrirá o navegador e começará a automatizar o cenário.

Seu primeiro cenário com capivara

O diagrama a seguir ilustra o fluxo do Pepino até o driver subjacente com a Capivara sentada no meio, atuando como tradutora:



Pepino e Selenium WebDriver são apenas joias adicionais. Para instalá-los, execute o seguinte:

```
gem install pepino Selenium-webdriver
```

[Se você estiver usando o Bundler, adicione pepino e selenium-webdriver ao seu Gemfile e execute o comando bundle install novamente.]

Nas versões do Capybara anteriores a 2.1, o Selenium WebDriver foi declarado como uma dependência de tempo de execução, caso em que teria sido instalado quando você instalou o Capybara e uma instalação separada não seria necessária.

Cucumber-Rails Se você

estiver usando o Capybara para testar uma aplicação Rails, você deve instalar a gem Cucumber-Rails em vez da gemRails padrão.

Esta gema tem tanto a Capivara quanto o Pepino declarados como dependências, então você os obterá gratuitamente quando instalar a gema. Para instalar o Cucumber-Rails, basta executar o seguinte comando:

```
gem install trilhos de pepino
```

Como alternativa, adicione isso ao seu Gemfile se estiver usando o Bundler, para que fique parecido com o seguinte:

```
fonte 'https://rubygems.org'
```

```
gema 'pepino-trilhos'
```

Seu primeiro cenário – uma pesquisa no YouTube

Agora que temos tudo o que precisamos, vamos decifrar e automatizar nosso primeiro cenário, uma simples pesquisa no YouTube:

Recurso: Pesquisar vídeos no YouTube

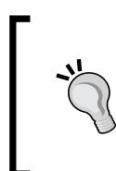
Cenário: Pesquisar vídeos de grandes roedores

Dado que estou na página inicial do YouTube

Quando procuro por "capivara"

Em seguida, vídeos de grandes roedores são retornados

Um guia completo para usar o Cucumber está fora do escopo deste livro, mas vamos supor que você tenha um arquivo/diretório configurado como o mostrado aqui e que possa executar recursos a partir da linha de comando usando o Cucumber.



Se você estiver usando o Bundler, execute o Cucumber usando o seguinte comando:
pacote de pepino executivo

Isso garantirá que você obtenha o executável Cucumber de seu pacote de gemas do projeto e não de nenhuma gema global.

```
features/ yy
youtube_search.feature yy step_defs yy steps.rb
```

```
yy
yy suporte yy
env.rb
```

Seu primeiro cenário com capivara

Supondo que você tenha um arquivo de recurso contendo o texto do cenário, ao executar Pepino na linha de comando, você deve obter os stubs de definição de etapa gerados:

pepino executivo pacote \$

Recurso: Pesquisar vídeos no YouTube

Cenário: Pesquisar vídeos de grandes roedores

Dado que estou na página inicial do YouTube

Quando procuro por "capivara"

Em seguida, vídeos de grandes roedores são retornados

1 cenário (1 indefinido)

3 etapas (3 indefinidas)

0m0.014s

Você pode implementar definições de etapas para etapas indefinidas com estes snippets:

Dado(/^estou na página inicial do YouTube\$/) do

pending # expressa o regexp acima com o código que você gostaria de ter

fim

When(/^I search for "(.*?)"\$/) do |arg1|

pending # expressa o regexp acima com o código que você gostaria de ter

fim

Então(/^vídeos de grandes roedores são retornados\$/) faça

pending # expressa o regexp acima com o código que você gostaria de ter

fim

Copie e cole os snippets gerados pelo Cucumber em seu arquivo steps.rb . Estes são os stubs que completaremos com nossos comandos Capivara.

Se você executar o Cucumber novamente, verá que ele informa que essas etapas existem, mas não foram implementadas:

pepino executivo pacote \$

Recurso: Pesquisar vídeos no YouTube

Cenário: Pesquisar vídeos de grandes roedores

Dado que estou na página inicial do YouTube

*Capítulo 1***TODO (Pepino::Pendente)****Quando procuro por "capivara"****Em seguida, vídeos de grandes roedores são retornados****1 cenário (1 pendente)****3 etapas (2 ignoradas, 1 pendente)****0m0.003s**

Agora temos código suficiente para trazer Capivara para a imagem. Começaremos adicionando a quantidade mínima de código necessária para iniciar a automação.

Verifique se o arquivo env.rb se parece com o seguinte:

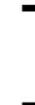
```
requerem 'capivara/pepino'
```

```
Capivara.default_driver = :selenium
```

Começamos adicionando require 'capivara/pepino'; isso é tudo que precisamos para carregar os arquivos necessários.



Em versões mais antigas do RubyGems, você pode precisar adicionar require 'rubygems' ao seu arquivo env.rb e se estiver usando o Bundler, você também precisará adicionar require 'bundler/setup'.



Em seguida, precisamos dizer ao Capivara para usar o driver Selenium usando:

```
Capivara.default_driver = :selenium
```

É importante reiterar novamente que o Capivara está simplesmente atuando como um tradutor e nos permite conversar com qualquer driver compatível. Neste caso, usamos o Selenium WebDriver porque é a ferramenta de automação de navegador de código aberto mais popular e nos permite testar em um navegador real, o que é útil para nosso primeiro teste na web.

Se você não definir o driver, poderá ver um erro como o seguinte:

\$ caixa/pepino

Recurso: Pesquisar vídeos no YouTube

Cenário: Pesquisar vídeos de grandes roedores

Dado que estou na página inicial do YouTube

**teste de rack requer um aplicativo de rack, mas nenhum foi fornecido
(ArgumentError)**

Quando procuro por "capivara"

Seu primeiro cenário com capivara

Em seguida, vídeos de grandes roedores são retornados

Cenários de falha:

pepino features/youtube_search.feature:3 # Cenário: Pesquisar vídeos de grandes roedores

1 cenário (1 falhou)

3 etapas (1 com falha, 2 ignoradas)

0m0.178s

Por padrão, o Capivara assume que você deseja testar um aplicativo Rack. O Rack é um middleware engenhoso usado nos frameworks Rails e Sinatra que permite testes full-stack da interação cliente/servidor sem a sobrecarga do HTTP, tornando os testes muito rápidos. Abordaremos isso em profundidade mais adiante no livro, quando discutirmos como o Capybara pode ser usado para testar aplicativos Rails e Sinatra.

Tudo o que resta agora é preencher as definições da etapa com nosso código Ruby que chamará a API Capivara para conduzir o teste.

Verifique se o arquivo steps.rb se parece com o seguinte:

```
Dado (/^estou na página inicial do YouTube$/) visite 'http://www.youtube.com'
```

```
fim
```

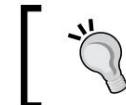
```
When(/^I search for "(.*?)"$/) do |search_term| fill_in 'search_query', :with =>
  search_term click_on 'search-btn' end
```

```
Então (/^vídeos de grandes roedores são retornados$/) do page.should have_content
  'Largest Rodent' end
```

Antes de dissecarmos o código, vamos rodar o teste e ver o que acontece. Como sempre, execute seu teste usando o comando pepino.

Espero que você veja o Firefox aberto, navegue até a página inicial do YouTube e pesquise vídeos de Capivara no YouTube.

Parabéns! Você acabou de executar com sucesso seu primeiro teste full-stack usando Capivara.



O único problema que posso ver que você pode ter aqui é se você não tiver o Firefox instalado ou se o tiver instalado em um local personalizado e o Selenium não conseguir encontrar o executável.

Vamos dar uma olhada rápida em cada etapa antes de nos aprofundarmos na rica API do Capivara. Aqui você verá como a API do Capivara é elegante, pois o código literalmente dispensa explicações. Esperançosamente, isso demonstra que, quando associado ao Cucumber, temos uma especificação legível por humanos, que é automatizada por um código muito expressivo. Para qualquer um que tenha vivido usando algumas das ferramentas de automação de teste disponíveis comercialmente, isso deve ser uma revelação!

A primeira linha diz ao Capybara para informar o driver (Selenium WebDriver) para abrir um navegador e navegar para um URL que fornecemos como string:

```
visite 'http://www.youtube.com'
```



O Selenium WebDriver possui mecanismos integrados para aguardar o carregamento de páginas no navegador, para que não tenhamos que nos preocupar com nenhum tipo de verificação de carregamento de página. Observe que isso não inclui a espera de JavaScript assíncrono, por exemplo, solicitações Ajax/XHTTP. Felizmente, a Capivara cobre isso, como descobriremos no devido tempo.

Depois disso, precisamos inserir nossos termos de pesquisa e clicar no botão **Pesquisar**. Então nós diga a Capivara para pedir ao motorista que preencha o formulário de busca com nossos termos de busca. Novamente, a API do Capivara é útil para nos dizer isso.

```
#observe que a variável search_term é passada do cenário Cucumber fill_in 'search_query', :with
=> search_term click_on 'search-btn'
```

A única fonte de confusão aqui pode ser o uso da string literal `search_query` no método `fill_in`. Muitos dos métodos do Capivara usam uma estratégia de "melhor palpite" quando você diz a eles para encontrar algo na página. Isso quer dizer que eles examinam vários atributos nos elementos DOM para tentar encontrar aquele que você pediu. Nesse caso, sabemos que o atributo de nome no elemento do formulário de pesquisa do YouTube é `search_query`, então foi isso que fornecemos.

Finalmente, precisamos verificar se os resultados retornados pela pesquisa foram relevantes. Para isso, usamos os matchers mágicos RSpec integrados do Capybara. Se você não sabe muito sobre RSpec, há muito online (<http://rspec.info/>), mas essencialmente os matchers fornecem maneiras semanticamente amigáveis de afirmar que o estado de algo é o que você espera, com a diferença das afirmações tradicionais sendo que eles geram exceções quando as condições não são atendidas (em vez de retornar `false`).

```
page.should have_content 'Maior Roedor'
```

Seu primeiro cenário com capivara

Por fim, vale a pena observar que o matcher `have_content` possui uma espera padrão incorporada. Isso é útil porque se o conteúdo que estamos esperando for carregado via JavaScript assíncrono (e não fizer parte do carregamento inicial da página), o Capivara tentará novamente por um período de tempo configurável para ver se existe. Abordaremos estratégias para lidar com JavaScript assíncrono posteriormente.

**Fazendo o download do código**

de exemplo Você pode fazer o download dos arquivos de código de exemplo para todos os livros Packt adquiridos em sua conta em <http://www.PacktPub.com>. Se você comprou este livro em outro lugar, visite <http://www.PacktPub.com/support> e registre-se para receber os arquivos por e-mail diretamente para você.

Resumo

O objetivo deste capítulo foi levá-lo a um ponto em que você automatizou seu primeiro cenário usando o Capivara. Verificamos se você tinha Ruby e RubyGems disponíveis e instalamos o Capybara e suas dependências, bem como o Cucumber como nosso executor de testes. Por fim, implementamos um cenário simples que automatizou uma pesquisa no YouTube; esperamos que isso tenha dado a você um vislumbre da elegante API do Capybara, que iremos investigar com mais detalhes no próximo capítulo.

2

Dominando a API

O exemplo de pesquisa do YouTube mostrou como é fácil automatizar cenários usando o Capivara. Agora precisamos abordar a API de frente, focando nos seguintes tópicos:

- Seletores – XPath ou CSS?
- Navegação •
Envio de formulários •
Localizadores, definição de escopo e várias
correspondências • Afirmação e consulta

No final deste capítulo, você deve se sentir confortável em automatizar seus próprios aplicativos usando o Capivara.

Localizando elementos com XPath e CSS

O Document Object Model (DOM) é uma estrutura "na memória" semelhante a uma árvore, que os navegadores constroem ao analisar uma página HTML e processar o JavaScript que existe embutido na página ou carregado por meio de tags de script . Os seletores CSS e as consultas XPath nos permitem pesquisar essa estrutura para encontrar conteúdo e o Capybara depende totalmente desses seletores para poder localizar o conteúdo nas páginas da web. Portanto, é essencial entendê-los antes de passar para a API.



Não deixe ninguém tentar dizer que XPath é mais rápido que CSS ou vice-versa. Capivara, na verdade, traduz todos os seletores CSS para XPath para que possamos fechar a tampa firmemente sobre aquela lata de vermes!

Dominando a API

Aqui estão alguns exemplos simples de cada tipo de seletor. Por exemplo, encontrar um elemento cujo atributo id tenha o valor 'main':

- XPath: //*[@id='main']
- CSS: #principal

Encontrando um filho direto ou indireto de qualquer elemento <div> com a classe 'container':

- XPath: //div//*[@class='container']
- CSS: div .container

O mais importante a considerar ao implementar um seletor é usar o menos frágil possível para recuperar o elemento desejado. Por exemplo, o seguinte seria um exemplo de uma expressão XPath muito ruim:

/html/body/div/div/div/div/p[1]

Qualquer alteração na estrutura da página provavelmente quebrará esse seletor, pois ele depende da hierarquia do DOM não mudar, o que não é uma expectativa razoável em nenhum aplicativo em desenvolvimento.

É muito melhor chegar a um elemento pela rota mais direta. Muitas vezes, isso significa usar uma combinação do tipo de elemento e um valor de atributo, por exemplo, id, classe ou nome.



Os aplicativos da Web projetados para serem acessíveis o ajudarão muito, porque geralmente usam padrões, como **WAI-ARIA** (<http://www.w3.org/WAI/intro/aria>) para adicionar atributos semânticos, significativos e consistentes aos elementos.

O tipo de seletor que você escolher depende inteiramente de você, embora eu me esforce para ser consistente em seus testes para ajudar na compreensão.

Vou usar seletores CSS ao longo deste livro, pois os considero um pouco mais legíveis do que o XPath.

Seletor padrão na Capivara

Capivara usa CSS como seletor padrão. Isso significa que ao usar a API você não precisará especificar qual seletor usar, como no exemplo a seguir:

```
page.find('#maincontent')
```

Capítulo 2

Se você deseja usar seletores XPath, você tem algumas opções. Em primeiro lugar, você pode declarar isso explicitamente ao chamar métodos:

```
page.find(:xpath, //*[@id="maincontent"])
```

Como alternativa, você pode configurá-lo globalmente. Este código provavelmente deve estar no seu arquivo env.rb se você estiver usando o Cucumber, como no exemplo a seguir:

```
requerem 'capivara/pepino'
```

```
Capivara.default_selector = :xpath
Capivara.default_driver = :selenium
```



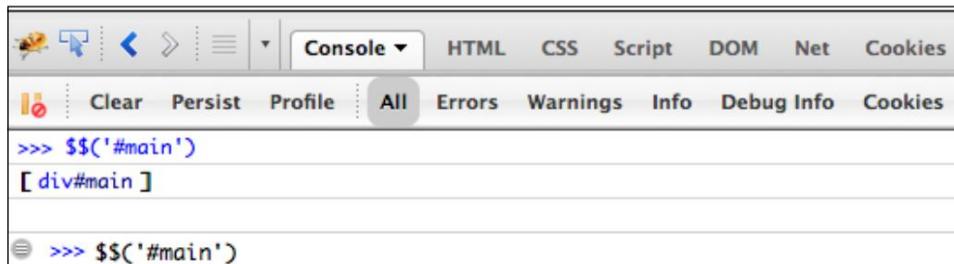
O Capybara suporta seletores CSS3, portanto, para todos os usuários avançados, sinta-se à vontade para usar :nth-child, :nth-of-type e todas as outras delícias que o CSS3 tem reservado.



Uma ajuda com os seletores

Existem algumas coisas realmente úteis que irão ajudá-lo se você não estiver confiante com os seletores XPath ou CSS.

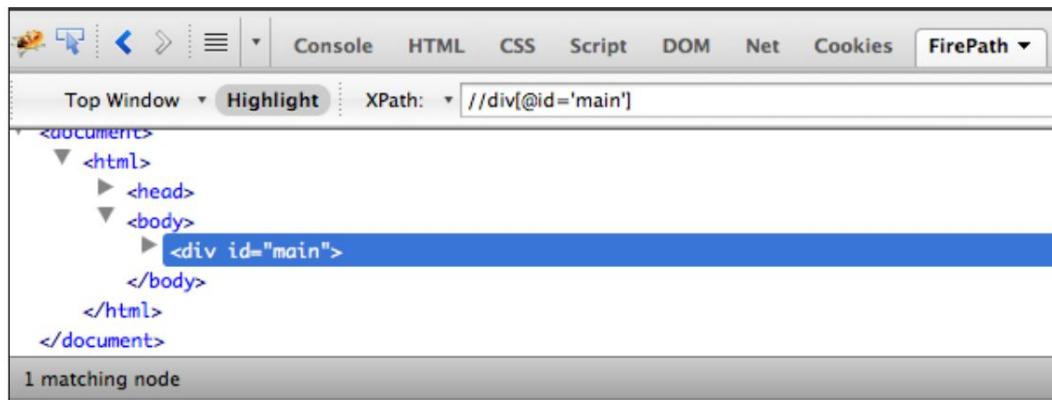
A primeira opção é que, nos navegadores Firefox e Chrome recentes, se você navegar para um console JavaScript, usando o Firebug no Firefox ou as ferramentas do desenvolvedor no Chrome, poderá experimentar seus seletores CSS e brincar até encontrar o melhor para o trabalho. Para fazer isso, basta usar um seletor CSS com um dólar duplo (\$\$) antes dele, como na captura de tela a seguir, que mostra um console do Firebug no Firefox. O mesmo se aplica às Ferramentas do Desenvolvedor no Chrome.



A segunda opção é instalar uma extensão do navegador, o que definitivamente vale a pena se você planeja usar o XPath como seu seletor padrão.

Dominando a API

Achei o complemento do Firefox **FirePath** (<https://addons.mozilla.org/en-us/firefox/addon/firepath/>) muito bom para essa finalidade, permitindo que você insira os seletores XPath ou CSS e destaque os correspondentes elementos no DOM.



Navegação

As primeiras partes da API Capivara com as quais você precisará se familiarizar são os métodos disponíveis para navegar em seu aplicativo.

Já usamos um desses métodos em nosso cenário de pesquisa no YouTube:

```
Dado(/^estou na página inicial do YouTube$/) visite 'http://
  www.youtube.com'
fim
```

Como seria de esperar, esse código faz com que o Capivara abra o navegador, se necessário, e navegue até o URL fornecido.

Clicar em links ou botões

Existem alguns métodos que podemos usar para navegar no aplicativo usando links ou botões:

- click_link_or_button
- click_on
- click_link

Aqui estão alguns exemplos de como poderíamos usá-los. A marcação fornecida é apenas um trecho de uma página da Web para que você possa ver como a API é usada no contexto:

```
<div id="principal">
    <div class="seção">
        <a id="myanchor" title="myanchortitle" href="#">Clique aqui
        Âncora</a> </
    div>
</div>
```

Qualquer uma das seguintes etapas do Cucumber clicaria com sucesso no link:

```
When(/^I click on a link using an id$/)
```

```
do click_on 'myanchor'
```

```
fim
```

```
When(/^I click on a link using text$/)
```

```
do click_link_or_button
```

```
'Click this Anchor'
```

```
fim
```

```
When(/^I click on a link using the title attribute$/)
```

```
do click_link 'myanchortitle'
```

```
fim
```

Nesses exemplos, usamos uma mistura dos três diferentes métodos de API disponíveis e também usamos diferentes seletores a cada vez.

Vimos no exemplo de "pesquisa do YouTube" que o Capivara geralmente usa uma estratégia de "melhor palpite" para grande parte da API ao tentar localizar elementos. No caso de links e botões, o Capivara verifica as seguintes propriedades do elemento ao tentar localizar o elemento para clicar:

- O atributo id da âncora, botão ou tag de entrada •
- O atributo title da âncora, botão ou tag de entrada •
- Texto dentro da âncora, botão ou tag de entrada •
- O atributo de valor do elemento de entrada onde seu tipo é um de 'botão', 'redefinir', 'enviar' ou 'imagem'
- O atributo alt onde uma imagem é usada como âncora ou entrada

Dominando a API

Podemos aplicar os mesmos métodos na seguinte marcação para clicar em um elemento de botão:

```
<div id="principal">
  <div class="seção">
    <button id="mybutton" title="mybuttontitle">Clique aqui
      Botão</button>
  </div>
</div>
```

When(/^I click on a button using a id\$/) do click_on 'mybutton'

fim

O mesmo é verdadeiro quando a marcação contém um elemento de entrada do tipo = "botão".

Envio de formulários

Outra tarefa comum que você provavelmente deseja automatizar é o preenchimento e o envio de formulários.

Mais uma vez, o Capybara fornece muitas APIs fáceis de usar para fazer exatamente isso. Considere este trecho de formulário simples:

```
<form id="myform"> <input
  type="text" name="Nome" value="" /> <input type="text"
  name="Sobrenome" value="" /> <input type="enviar" valor="Ir" />

</form>
```

A seguinte definição da etapa Cucumber preencheria e enviaria o formulário:

```
Quando (/^eu preencher e enviar o formulário$/) preencha_em
  'Nome', :com => 'Mateus' preencha_em 'Sobrenome', :com =>
  'Robbins' clique_em 'Ir'
```

fim

Ao localizar campos que podem aceitar entrada de texto, o Capybara usará um dos seguintes para localizar esses campos no DOM:

- O atributo id do elemento de entrada
- O atributo de nome do elemento de entrada
- Um elemento de rótulo relacionado

Um exemplo usando elementos de rótulo é mostrado no código a seguir. Os rótulos são mais comumente associados a botões de opção e caixas de seleção, mas ainda podem ser usados com entradas de texto:

```
<form id="myform"> <label  
for="name1">Nome do usuário</label> <input id="name1"  
type="text" name="Forename" value="" /> <label for=" name2">Sobrenome do usuário</  
label> <input id="name2" type="text" name="Sobrenome" value="" /> <input type="submit"  
value="Go" /> </form>
```

Quando(/^preencher e enviar o formulário\$/) faça
 fill_in 'Nome do usuário', :with => 'Mateus' fill_in 'Sobrenome do
 usuário', :with => 'Robbins' click_on 'Ir' end

Caixas de seleção e botões de opção

Também é provável que você encontre formulários com caixas de seleção e botões de opção. Felizmente, a API para manipular esses elementos é muito parecida com a conclusão de entradas de texto.

A seguinte marcação agora inclui alguns elementos adicionais e uma captura de tela é mostrada para que você possa visualizar como ela ficaria quando renderizada em um navegador:

```
<form id="myform"> <label  
for="name1">Nome do usuário</label> <input id="name1"  
type="text" name="Forename" value="" /> <label for=" name2">Sobrenome do usuário</  
label> <input id="name2" type="text" name="Sobrenome" value="" />  
  
<p>  
  <label for="title">Título</label> <select  
  name="user_title" id="title">  
    <option>Sra</option>  
    <option>Sr</option>  
    <option>Sra</option> </select>  
</p>  
  
<p>  
  <label for="under_16">Abaixo de 16</label> <input  
  type="radio" name="underage" value="under"  
  id="under_16" checked="checked"/>
```

Dominando a API

```

<label for="over_16">Acima de 16</label> <input
type="radio" name="overage" value="over" id="over_16"/>

</p>
<p>
    <label for="consent">Consentimento concedido?</label> <input
    type="checkbox" value="yes" name="consent_checkbox" id="consent"/>

</p>
<input type="submit" value="Go" />
</form>

```

Isso gerará uma saída conforme mostrado na captura de tela a seguir:

Agora precisamos implementar etapas para manipular o formulário usando os menus suspensos, botões de opção e caixas de seleção. Estes são apenas elementos de entrada do tipo select, radio e checkbox , respectivamente.

```

Quando(/^preencher e enviar o formulário$/) faça
    fill_in 'Nome do usuário', :with => 'Mateus' fill_in 'Sobrenome do
    usuário', :with => 'Robbins' selecione 'Sr.', :from => 'título'

    escolha 'acima de 16 anos'
    marque 'consentimento'
    click_on 'Ir' fim

```

Já cobrimos o preenchimento dos campos de texto. Em seguida, passamos a selecionar o título na lista suspensa:

```
selecione 'Sr', :from => 'título'
```

Capítulo 2

Tal como acontece com os elementos de entrada , o Capybara examinará novamente os rótulos relacionados, os atributos id e name , para localizar o elemento. Nesta instância, o título é o ID do elemento selecionado . O valor a ser selecionado na lista deve ser o texto de um dos elementos de opção filho .



Capivara também possui um método de desmarcar, que faz exatamente o que você esperaria; limpa a seleção!



O próximo elemento a ser abordado é a seleção do botão de opção. Aqui usamos o seguinte código:

```
escolha 'acima de 16 anos'
```

O método de escolha do Capivara novamente examina os rótulos relacionados, os atributos id e name , para localizar o elemento. Neste caso, Over 16 é o rótulo relacionado à entrada do rádio com o valor id como over_16.

Por fim, precisamos marcar a caixa de seleção Consentimento e para isso usamos:

```
marque 'consentimento'
```

O método check do Capivara , assim como os demais, utiliza rótulos relacionados, os atributos id e name para localizar o elemento. Nesta instância, o consentimento é o valor de id relacionado à entrada da caixa de seleção.



Capivara também possui um método de desmarcação, que faz exatamente o que você esperaria; ele limpa a caixa de seleção!



Por fim, o Capybara também fornece API para permitir que você carregue arquivos definindo o caminho em um elemento de entrada de type='file'. Aqui está um exemplo de marcação contendo tais um elemento:

```
<label for="form_image">Imagem</label> <input type="file"
name="image" id="form_image"/>
```

E uma etapa de exemplo para implementar isso seria a seguinte:

```
Quando(/^anexer um arquivo em um form$/) faça
  attach_file 'Imagen', '/Users/matt/foo.png' fim
```

Dominando a API

O Capivara validará se o arquivo para o qual você está tentando definir o caminho realmente existe no sistema de arquivos. Caso contrário, você verá uma mensagem de erro como a seguinte:

não é possível anexar arquivo, /Users/matt/foo.png não existe

(Capivara::FileNotFoundException)

./features/step_defs/form.rb:11:

Localizadores, escopo e várias correspondências

Grande parte da API que abordamos até agora neste capítulo foi, de certa forma, uma fachada.

Capivara, da melhor maneira possível, fornece muito "açúcar sintático" em torno de alguns blocos básicos de construção.

Esses blocos de construção são, na verdade, simplesmente expressões XPath para encontrar coisas na página e, em seguida, delegar a ação ao driver subjacente.

Na maioria das vezes, faz sentido usar essa API "açucarada", pois seu código fica muito mais expressivo e legível. Além do benefício óbvio de "escrever uma vez, executar em vários drivers", a semântica limpa da API do Capivara é seu principal ponto de venda, portanto, você deve usá-la sempre que possível.

No entanto, haverá momentos em que esses métodos não funcionarão para você. Por exemplo, o método `click_on` é ótimo para lidar com a navegação por meio de tags âncora e imagens, mas se o seu site usar muito JavaScript para registrar eventos de clique em outros elementos?

Considere a seguinte página da web (bastante inútil):

```
<html>
<cabeça>
<title>Clique em Exemplos</title> <script>
window.onload = function() {

    var mydiv = document.getElementById("mydiv");

    mydiv.onclick =function() { alert('div foi
        clicado');
    };

}</script> </
head> <body>
```

Capítulo 2

```
<div id="principal">
  <div class="seção">
    <div id="mydiv" title="mydivtitle">Clique neste Div</div> </div> </body> </html>
```

Nesse caso, clicar no elemento div com o valor id de mydiv resulta em uma mensagem de alerta.

Para automatizar esse clique, precisamos recorrer aos chamados "finders" da Capivara. Para começar, vamos focar nossa atenção no próprio método `find`. Este é um método que pega uma expressão XPath ou um seletor CSS e retorna um `Capybara::Element` no qual podemos invocar uma ação.

Uma etapa do Cucumber para fazer isso seria algo como isto:

```
Quando(/^eu clico em um div com um manipulador de cliques anexado$/) faça
  find('#mydiv').clique
fim
```

Como escolhemos seletores CSS como seletor padrão, não precisamos informar isso ao Capivara; no entanto, se quiséssemos usar um XPath, simplesmente passaríamos `:xpath` como o primeiro argumento para o método e, em seguida, a expressão XPath como o segundo argumento.

Quase todos os outros métodos do localizador são construídos em cima disso, mas são um pouco mais açucarados:

- `find_field`: Este localizador procura campos de formulário pelo elemento `label` relacionado ou pelo atributo `name/id`
- `field_labeled`: Este localizador é o mesmo que `find_field` • `find_link`: Este localizador encontra uma âncora ou um link de imagem usando o atributo `text`, `id` ou `img alt`
- `find_button`: Este localizador encontra um botão pelo atributo `id`, `nome` ou `valor` • `find_by_id`: Este localizador encontra qualquer elemento pelo atributo `id`

Como você pode ver, os localizadores são extremamente poderosos, permitindo-nos localizar qualquer elemento no DOM e manipulá-lo.

Múltiplas correspondências

Até agora, todos os exemplos que vimos assumem que estamos procurando por um único elemento no DOM. Com aplicativos da Web dinâmicos, você pode não saber exatamente qual elemento procurar. Por exemplo, vamos supor que você esteja verificando alguns resultados de pesquisa que são adicionados dinamicamente à página:

```
<div id="principal">
  <h1>Resultados da pesquisa</h1>
  <ul class="seção">
    <li id="res1" class="result">Correspondência 1</li> <li id="res2"
      class="result">Correspondência 2</li> <li id="res3" class="result"
      >Correspondência 3</li> </ul> </div>
```

Há algumas coisas que você pode fazer aqui para afirmar que o conteúdo que você espera que seja retornado existe. O mais óbvio e talvez aquele para o qual você pode recorrer primeiro é iterar os resultados e inspecionar cada um. Para isso você pode usar o método `all`, que retorna uma coleção de elementos:

```
Quando(/^procuro o resultado relevante$/) faço
  all('.result').each_with_index do |elem, idx|
    elem.text.should ==
      "Correspondência #{idx + 1}"
  fim
fim
```

Aqui estamos iterando por todos os resultados da pesquisa na marcação anterior e verificando se o texto é o que esperamos, ou seja, o primeiro elemento contém Correspondência 1, o segundo Correspondência 2 e assim por diante.

Existem algumas alternativas para usar o método `all`, que podem ser mais apropriadas. Se você não precisar verificar cada correspondência, poderá definir sua estratégia de correspondência para ser mais inteligente, talvez procurando correspondências parciais ou tentar passar alguns argumentos adicionais para garantir que o elemento esteja visível e contenha um texto específico. Após o lançamento do Capivara 2.1, houve uma mudança significativa nessa área, por isso faz sentido cobrir isso em profundidade.

Estratégias de correspondência

Olhando novamente para a marcação de resultados de pesquisa anterior, vamos ver o que acontece quando você usa o método `find` para recuperar elementos com o resultado da classe:

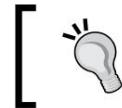
```
Quando(/^procuro o resultado relevante$/) faço
  find('.resultado').text
fim
```

A execução desse código produz o seguinte erro:

```
Quando procuro o resultado relevante          # features/step_defs/
resultados.rb:5

Correspondência ambígua, encontrados 3 elementos correspondentes a css ".result"
(Capivara::Ambíguo)
```

Capivara está informando que há mais de um elemento correspondente a essa consulta, o que faz sentido. Não queremos que a Capivara seja muito mágica e apenas assumamos que queremos a primeira partida; isso pode ser perigoso.



O Capybara 1.0 pegaria apenas a primeira de várias correspondências, o que poderia resultar em seu código pegando o elemento incorreto e causando dores de cabeça ao depurar testes quebrados.

Capybara 2.1 introduziu as opções Capybara.match e Capybara.exact para permitir que você ajuste a estratégia empregada ao tentar encontrar elementos na página. As opções possíveis para Capivara.match são:

- :one – Esta opção irá gerar uma exceção Capybara::Ambiguous quando mais de uma correspondência for encontrada para a consulta.
- :first – Esta opção simplesmente escolherá a primeira correspondência (o antigo comportamento).
- :prefer_exact – Esta opção retornará um elemento de correspondência exata, se várias correspondências forem encontradas, algumas das quais são exatas e outras não.
- :smart – Esta opção está disponível por padrão no Capivara 2.1 e é dependente do valor de Capivara.exata. Se for definido como true, o comportamento é o mesmo que :one. Caso contrário, o Capybara primeiro procura correspondências exatas, se várias correspondências forem encontradas, uma exceção Capybara::Ambiguous é levantada, se nenhuma for encontrada, ele procura correspondências inexatas novamente levantando Capybara::Ambiguous se várias correspondências forem retornadas.

Aqui está um exemplo para ilustrar as diferentes opções. Esperamos que você não tenha uma marcação como esta em seu aplicativo, mas serve para ilustrar como as diferentes estratégias afetarão quais elementos são retornados de uma consulta.

```
<label for="text1">preencha este campo</label> <input id="text1" type="text"/>
<label for="text2">preencha isto</label> <input id="text2" type="text" /> <label
for="text3">preencha esta entrada</label> <input id="text3" type="text" />
```

```
page.fill_in 'por favor complete isto', :with => 'foobar' page.fill_in 'por favor
complete', :with => 'bazqux'
```

Dominando a API

Lembre-se, o comportamento padrão quando não estiver usando seletores XPath ou CSS é que o Capivara tentará executar uma correspondência parcial de string. No exemplo anterior, o primeiro termo de pesquisa preencha esta é uma correspondência parcial para os rótulos preencha este campo e preencha esta entrada. O segundo termo de pesquisa, por favor, complete é uma correspondência parcial para todos os três rótulos.

Vamos considerar cada estratégia por vez e ver como isso afetaria qual entrada é concluída e com qual valor:

- Capybara.match = :one – Esta estratégia levanta o Capybara::Ambiguous exceção.
- Capybara.match = :first – Nesta estratégia, ambas as chamadas fill_in preencherão a primeira entrada de texto, pois ambas são correspondências parciais para o primeiro rótulo.
- Capybara.match = :prefer_exact – Nesta estratégia, a segunda entrada será completada com foobar , pois seu rótulo é uma correspondência exata para a primeira consulta. Capivara voltará a preencher bazqux para a primeira entrada, pois seu rótulo é a **primeira** correspondência parcial para a segunda consulta. Observe que esta consulta também é uma correspondência parcial para a segunda e terceira entrada, mas com essa estratégia, a primeira de todas as correspondências múltiplas é usada.
- Capivara.match = :smart – Nesta estratégia, a segunda entrada será preenchido com foobar, seu rótulo é uma correspondência exata para a primeira consulta. No entanto, ao contrário de :prefer_exact a Capybara::Ambiguous, um erro é gerado para a segunda chamada fill_in porque há mais de uma correspondência parcial.



Observe que se você definir Capybara.exact = :true, isso substituirá as estratégias anteriores e nenhuma correspondência parcial será considerada. Em vez disso, a exceção Capybara::ElementNotFound será gerada se uma consulta corresponder apenas parcialmente aos elementos na página.

O comportamento de :smart também se estende à escolha de opções em uma lista suspensa (o elemento select).

Como você pode ver, essas estratégias são muito difíceis de entender, então eu recomendo que você crie algumas de suas próprias páginas de teste e, em seguida, experimente as diferentes opções até se sentir confortável com o que funcionará melhor em sua produção testes.

Visibilidade do elemento

A visibilidade de um elemento também afeta se a Capivara irá localizá-lo no DOM.

Capivara tem uma configuração global, que usa para determinar se deve verificar se há elementos ocultos:

```
Capivara.ignore_hidden_elements = verdadeiro
```

O valor padrão para esta configuração agora é verdadeiro, o que significa que os elementos que não são visíveis para o usuário (por exemplo, eles têm propriedades CSS definidas como `display: none` ou `visibilidade: oculto`) não serão retornados como resultados de nossas consultas .

O comportamento do método de texto também mudou um pouco no Capivara 2.1. Isso é normalmente usado da seguinte forma:

```
find('#log').text #encontra o texto dentro do elemento com o id 'log'
```

O comportamento deste método agora depende do valor de `Capybara.ignore_hidden_elements`. Quando isso é verdade, apenas o texto visível é retornado, caso contrário, todo o texto é devolvido.



No Capybara 1.0 o comportamento deste método era dependente do driver; O Selenium WebDriver retornou apenas texto visível, mas Rack::Test retornaria qualquer coisa. No Capivara 2.0, a consistência foi aplicada e ambos os drivers retornaram apenas texto visível.

O Capybara 2.1 também introduziu um método para substituir o comportamento do Capybara. `ignore_hidden_elements` para o método de texto . Se você definir `Capybara.visible_text_only = true`, o comportamento do método de texto será retornar apenas texto visível, independentemente de `ignore_hidden_elements` estar definido como false.



É importante lembrar que o comportamento do motorista ainda pode diferir dependendo de quão sofisticada é a interpretação do motorista sobre a visibilidade. No Rack::Test, isso será bastante grosseiro, pois não há renderização, ao contrário do Selenium, onde as páginas são totalmente renderizadas. Portanto, o navegador pode aplicar uma interpretação mais sofisticada sobre se um elemento está visível para o usuário.

Por fim, vale a pena notar que você pode definir todas essas opções em um bloco de configuração.

Por exemplo:

```
Capivara.configure do |config|
  config.match
  = :smart
  config.exact_options = true
end
```

Escopo

Ao tentar localizar conteúdo na página que pode não ser fácil de identificar exclusivamente, outra opção é restringir a consulta a uma seção da página. Capybara fornece o método `inside` para permitir consultas com escopo:

```
<div id="principal">
  <h1>Resultados da pesquisa</h1>
  <ul id="local_results">
    <li id="res1" class="result">Correspondência local 1</li> <li id="res2"
      class="result">Correspondência local 2</li> <li id="res3" class=""
      resultado">Correspondência local 3</li> </ul> <ul id="internet_results">

    <li id="res4" class="result">Correspondência 1 na Internet</li> <li id="res5"
      class="result">Correspondência 2 na Internet</li> <li id="res6" class=""
      resultado">Internet Match 3</li> </ul> </div>
```

No exemplo anterior, temos dois conjuntos de resultados de pesquisa retornados, alguns do site local e outros da Internet. Vamos supor que queremos encontrar todos os resultados da Internet.

```
When(/^I search for results in a scope$/) do within('#internet_results') do
  all('.result').each do |elem|
    coloca elemento.texto
  fim
  fim
  fim
```

A execução desta etapa produziria a seguinte saída:

```
Internet Correspondência 1
Internet Correspondência 2
Internet Match 3
```

O argumento para o método `inside` é exatamente o mesmo de `find`. Se nenhum tipo for fornecido, o único argumento será considerado um seletor do tipo padrão.

O Capybara também fornece alguns métodos internos que abrangem um tipo específico de elemento:

- `within_fieldset` – O primeiro argumento deve ser o atributo `id` ou `legend` dentro de um elemento `fieldset` de formulário
- `within_table` – O primeiro argumento deve ser o atributo `id` ou `caption` dentro de um elemento de tabela

- `within_frame(frame_id)` – O primeiro argumento deve ser o valor id de um elemento iframe (drivers selecionados, por exemplo, Selenium)
- `within_window` – O primeiro argumento deve ser o identificador da janela (drivers selecionados, por exemplo, Selenium)

Afirmando e consultando

Agora que você pode navegar em seu aplicativo, enviar formulários e localizar qualquer elemento no DOM, precisamos voltar nossa atenção para validar o comportamento esperado.

A capivara nos permite fazer isso de duas maneiras.

A primeira opção é usar a API Capivara "Query" diretamente. Capivara fornece todo um conjunto de métodos para consultar a página em teste e retornar um valor booleano.

```
page.has_content? página 'pedras de
capivara'.has_selector? '#principal'
```

Você pode usar esses métodos com uma abordagem tradicional de "asserção", em que os testes afirmam contra uma condição booleana. A outra opção é usar RSpec "Magic Matchers". Esses matchers, na verdade, apenas "pegam carona" nos métodos de consulta:

```
page.should have_content 'capivara rocks' page.should
have_selector '#main'
```

A beleza de usar os matchers RSpec é dupla:

- Você obtém exceções significativas que informam exatamente onde está o problema
- Satisfaz a ideologia de "falha rápida", segundo a qual falhamos o mais cedo e o mais forte possível, exatamente o que queremos ao testar aplicativos da web

Matchers e RSpec

Capivara expõe os seguintes métodos para consultar as páginas do seu aplicativo:

```
has_selector?(*args)
has_no_selector?(*args) has_xpath?
(caminho, opções={}) has_no_xpath?
(caminho, opções={}) has_css?(caminho,
opções={}) has_no_css?(caminho, opções={ })
has_text?(conteúdo) has_content?(conteúdo)
```

Dominando a API

```

has_no_text?(conteúdo)
has_no_content?(conteúdo)
has_link?(localizador, opções={}) has_no_link?
(localizador, opções={}) has_button?(localizador)
has_no_button?(localizador) has_field?(localizador,
opções={}) has_no_field?(locator, options={})
has_checked_field?(locator) has_no_checked_field?
(locator) has_unchecked_field?(locator)
has_no_unchecked_field?(locator) has_select?
(locator, options={}) has_no_select?(locator,
options={}) has_table?(localizador, opções={})
has_no_table?(localizador, opções={})

```

Todos eles podem ser usados isoladamente ou agrupados usando um RSpec Matcher.

Não entraremos em muitos detalhes sobre esses métodos de consulta porque os princípios são exatamente os mesmos de quando usamos os métodos "finder"; a única diferença é que, em vez de retornar um elemento, eles apenas retornam true ou false, dependendo se o elemento ou o conteúdo existe.

Voltemos ao nosso exemplo de "resultados da pesquisa":

```

<div id="principal">
  <h1>Resultados da pesquisa</h1>
  <ul id="local_results">
    <li id="res1" class="result">Correspondência local 1</li> <li id="res2"
      class="result">Correspondência local 2</li> <li id="res3" class="resultado">Correspondência local 3</li> </ul> <ul id="internet_results">
    <li id="res4" class="result">Correspondência 1 na Internet</li> <li id="res5"
      class="result">Correspondência 2 na Internet</li> <li id="res6" class="resultado">Internet Match 3</li> </ul> </div>

```

Aqui estão alguns exemplos de consultas e seus equivalentes RSpec, primeiro verificando a existência de um seletor e, em seguida, verificando o conteúdo do texto:

```

Então(/^os resultados de pesquisa desejados são retornados$/) faça
  page.has_selector? '#local_results' page.should
  have_selector '#local_results'

```

```
fim
Então(/^os resultados de pesquisa desejados são retornados$/) faça
  page.should have_content 'Local Match 1' first('#res1').should
    have_content 'Local Match 1'
fim
```

Observe que esses métodos de consulta podem ser usados no nível da página ou em qualquer nó recuperado usando um método localizador.



Se você vir a exceção undefined method should for #<Capybara::Session> (NoMethodError), provavelmente precisará adicionar gem install rspec ou adicionar gem 'rspec' ao seu Gemfile e executar a instalação do bundle novamente. Você também precisaria adicionar require 'rspec/expectations' em seu arquivo env.rb.

Além de verificar se os elementos e o conteúdo existem, é claro que podemos verificar se as coisas não estão presentes ou visíveis na página. No exemplo a seguir, verificaremos se um elemento com o valor id local_results não existe na página:

```
Então (/^os resultados de pesquisa desejados são retornados$/) do
  page.has_no_selector? '#local_results' page.should have_no_selector
    '#local_results'
fim
```

O has_xpath?, has_no_xpath?, has_css? e has_no_css? métodos são precisamente os mesmos que o has_selector? e has_no_selector? métodos, a única diferença é que eles usam especificamente seletores XPath ou CSS como argumentos.

Os métodos que examinam elementos específicos, como tabelas, campos e links, se comportam exatamente como os métodos localizadores que vimos anteriormente, onde aceitam um argumento localizador e examinam diferentes atributos ou rótulos para localizar esses elementos.

- has_link? – Isso verifica uma âncora ou um link de imagem usando o atributo text, id ou img alt
- has_button? – Isso verifica um botão pelo atributo id, name ou value
- has_field? – Isso verifica o atributo rótulo, nome ou id associado de um campo
- has_select? – Isso verifica o atributo label, name ou id associado de uma entrada do tipo select
- has_table? – Isso verifica o elemento id ou caption dentro de um elemento de tabela



É importante lembrar que todos esses métodos podem ser chamados em qualquer elemento Capybara::Node, ou seja, você pode realizar uma consulta em um nó que foi retornado a partir do resultado de uma consulta anterior.

Refinando localizadores e compensadores

Além de definir a configuração do Capybara globalmente usando opções como Capybara.match = :smart, você pode substituir o comportamento em instruções de localizador único ou correspondente passando um hash adicional de argumentos, como :text, :visible, :exact, :match, ou :espere.

Considere este trecho de página, que torna algum texto visível após um atraso de cinco segundos:

```
<cabeça>
<script> $ 
  (document).ready(function() { var addText =
    function() {
      $('.section').attr('estilo', 'visibilidade;visível;');
      
    } setTimeout(addText, 5000);
  });
</script> </
head>
<corpo>
<div id="principal">
  <div class="section" style="visibility:hidden;">Capivara
  Rochas</div> </
div> </body>
```

Se não tivermos modificado Capybara.default_timeout, não ajudará aqui, pois o padrão é aguardar dois segundos para JavaScript assíncrono. Podemos substituir isso e, enquanto estamos nisso, fazer nosso localizador verificar o texto correto, bem como substituir a estratégia de correspondência e garantir que o texto esteja visível!

```
find('.section', :visible => true, :wait => 10, :text => 'Capivara R', :match => :first)
```

Vale a pena afirmar novamente que isso substituirá quaisquer configurações padrão que possamos ter. Essa técnica também pode ser usada com os métodos de consulta (como page.has_content?) e também com o método all .

Capítulo 2

Por exemplo, vamos supor que uma página executou algum JavaScript que definiu um dos resultados da pesquisa para não ser exibido:

```
<div id="principal">
    <h1>Resultados da pesquisa</h1>
    <ul class="seção">
        <li id="res1" class="result">Correspondência 1</li> <li id="res2"
            class="result">Correspondência 2</li> <li id="res3" class="result"
            >Combinar 3</li> <li id="res4" class="result"
            style="visibility:hidden;">Combine
            4</li>
        <li id="res5" class="result">Correspondência 5</li> </ul> </div>
```

Se agora executarmos o seguinte passo:

```
Quando(/'procuro por resultados visíveis') faço
    all('.resultado', :visible => true).each do |elem|
        coloca elemento.texto
    fim
fim
```

A saída seria:

```
Correspondência 1
Combinar 2
Combinar 3
Combinar 5
```

Você também pode obter o método all para retornar apenas elementos com texto específico, por exemplo, se modificarmos a etapa novamente:

```
Quando(/'procuro o resultado relevante') faço
    all('.resultado', :texto => 'Correspondência 1').each do |elem| coloca
        elemento.texto
    fim
fim
```

Então a saída seria:

```
Correspondência 1
```

Verificando os valores dos atributos

Muitas vezes você vai querer verificar se um elemento existe e tem um atributo com um valor específico. Isso é simples usando seletores CSS ou expressões XPath.

Este exemplo verifica a presença de um elemento div com um atributo id de local_results:

```
Então(/^os resultados de pesquisa desejados são retornados$/) faça
  #usando o seletor CSS
  page.has_selector? 'div[id=local_results]' #usando XPath
  page.has_xpath? "//div[@id='local_results']" fim
```

Você também pode usar uma notação de índice de matriz em qualquer Capivara::Elemento passando o nome do atributo como um símbolo, conforme mostrado no exemplo a seguir. Aqui combinamos essa técnica com um dos matchers padrão do RSpec para afirmar o resultado esperado.

```
Então(/^os resultados de pesquisa desejados são retornados$/) faça
  first('#res1')[:class].should == 'resultado'
fim
```

Resumo

Neste capítulo, fizemos um tour guiado pela API do Capivara, abrangendo navegação, preenchimento de formulários, localização de elementos e validação de seu conteúdo. A boa notícia é que a parte mais difícil já está resolvida e agora você tem todas as habilidades necessárias para automatizar seus testes com o Capivara. No próximo capítulo, veremos como o Capivara é poderoso para testar aplicativos Rails e Sinatra e começaremos a descobrir os benefícios reais de usar essa biblioteca maravilhosa.

3

Testando Rails e Sinatra Formulários

O Capybara nasceu do ecossistema de ferramentas que existem para dar suporte a testes de aplicações Rails. Vimos que qualquer aplicação web pode ser testada usando Capybara; no entanto, devemos dedicar um tempo para entender por que o Capybara é particularmente adequado para testar aplicativos Rails, aplicativos Sinatra ou, na verdade, qualquer aplicativo Rack.

Neste capítulo vamos considerar os seguintes tópicos:

- Definindo Rack •
- Capivara e Rack::Test • Qual driver usar e quando
- Capivara, Rails e acessórios transacionais

Entendendo a interface do Rack

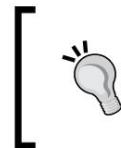
Se você ficar tempo suficiente em qualquer quadro de mensagens Ruby, canal IRC (Internet Relay Chat) ou Issue Tracker, logo ouvirá menções ao misterioso aplicativo Rack; Então, o que é Rack?

Rack (<http://rack.github.io/>) é provavelmente uma das bibliotecas mais poderosas dentro da pilha de aplicativos web Ruby e sustenta quase todas as estruturas e servidores web baseados em Ruby. Rack é uma camada de abstração, geralmente chamada de middleware, situada entre qualquer aplicativo da Web que deseja se comunicar por HTTP e o servidor da Web escolhido para implementar essa comunicação.

Testando aplicativos Rails e Sinatra

O que isso significa na prática? Digamos que você esteja implementando uma estrutura de aplicativo da web. Em algum momento, você terá que considerar como fará a interface com o servidor da Web para poder digerir uma solicitação e enviar uma resposta. Normalmente, isso teria que ser feito gravando em um adaptador sob medida para o servidor da Web fornecido. Portanto, se você deseja que sua estrutura da Web seja executada em vários servidores, terá que codificar vários adaptadores. Rack elimina a necessidade disso; você simplesmente codifica sua estrutura para a interface muito direta do Rack, e o Rack lida com a comunicação com o servidor, portanto, a troca de servidores é uma alteração de uma linha!

O seguinte é provavelmente o exemplo de trabalho mais simples de um aplicativo Rack. Experimente você mesmo e verá que não há mágica aqui, apenas uma ideia muito simples, mas engenhosa.



Você precisará da gema Rack para isso. Você deve tê-lo em seu projeto atual como uma dependência do Capivara; mas se você tiver algum problema, certifique-se de executar `gem install rack` ou adicionar `gem 'rack'` ao seu Gemfile e executar `bundle install`.

```

exigir 'rack'
servidor'

classe HelloWorld
resposta definitiva
[200, {'Content-Length' => '11'}, ['Hello World']]
fim
fim

classe HelloWorldApp
def self.call(env)
  HelloWorld.new.response
fim
fim

Rack::Server.start :app => HelloWorldApp

```

Este exemplo implementa a chamada do método Rack principal e, em seguida, responde com um status simples de 200 OK e o corpo do texto Hello World. Note que ao invés de usar `Rack::Server`, que usa o servidor WEBrick padrão, poderíamos ter usado qualquer servidor web compatível com Rack aqui. Trocar isso por outro servidor como o Thin (<http://code.macournoyer.com/thin/>) seria tão simples quanto mudar a última linha para `Rack::Handler::Thin.start`.

Você pode iniciar o aplicativo digitando ruby myapp.rb na linha de comando; você deve então ver o seguinte exibido:

```
$ ruby meuaplicativo.rb
[2013-04-09 08:29:27] INFO WEBrick 1.3.1
[2013-04-09 08:29:27] INFO ruby 1.9.3 (2012-11-10) [x86_64-darwin11.4.2]
[2013-04-09 08:29:27] INFO WEBrick::HTTPServer#start: pid=32991 port=8080
```

Se você navegar para http://localhost:8080, deverá ver o texto Hello World no navegador, pronto para seu próprio aplicativo Rack!

Os benefícios se estendem além dessa abstração, porque a comunicação entre a estrutura da web e o servidor da web é feita por meio do formato de hash especificado do Rack, o que significa que a eliminação de solicitações e respostas é direta. A biblioteca de teste só precisa estar em conformidade com o formato de hash para solicitações e manipulação de resposta, e você pode exercitar instantaneamente a funcionalidade de pilha completa do aplicativo sem a necessidade de um servidor web e HTTP. A biblioteca Rack::Test fornece essa funcionalidade e é um dos drivers integrados fornecidos com o Capybara. É provável que seja seu primeiro porto de escala ao testar um aplicativo Rails ou Sinatra com o Capybara.

Capivara e Rack::Teste

Rack::Teste é uma biblioteca que implementa o protocolo Rack, possibilitando testar a funcionalidade full-stack da sua aplicação sem a latência da comunicação HTTP e abrir navegadores reais. Você pode usar sua API para enviar uma solicitação ao seu aplicativo; quando seu aplicativo responder, Rack::Teste digerirá a resposta e permitirá que você a interroge. Ele pode fazer isso porque está simplesmente implementando o protocolo Rack, construindo o hash de solicitação para apresentar ao aplicativo da mesma forma que o servidor web Rack faria, interpretando o hash de resposta e disponibilizando-o para você por meio da API.

Aqui está um exemplo usando Rack::Teste fora do Capivara que testa o aplicativo "Hello World" que desenvolvemos anteriormente:

```
exigir 'rack/teste' exigir 'teste/
unidade'

class HelloWorldAppTest < Test::Unit::TestCase
  include Rack::Teste::Métodos
```

aplicativo de definição

Testando aplicativos Rails e Sinatra

```

HelloWorldApp
fim

def test_redirect_logged_in_users_to_dashboard
    pegar "/"

    afirmar last_response.ok?
    assert_equal last_response.body, 'Hello World'
fim

fim

```

Esse teste chama a rota da página inicial de nosso aplicativo, afirma que um código de resposta HTTP 200 foi recebido e, em seguida, garante que o texto do corpo seja Hello World. Mas o mais importante a lembrar é que nenhuma comunicação real ocorreu aqui, portanto, você obtém a confiabilidade e a velocidade de um teste de unidade com o benefício de exercitar a pilha completa do aplicativo.

É claro que, realisticamente, nosso aplicativo em teste provavelmente será implementado usando estruturas como Rails ou Sinatra; também queremos usar o Capybara para conduzir o Rack::Test, então não precisamos nos preocupar com as partes internas do Rack::Test.

Rack::Test não possui uma API direta para clicar em elementos, definir botões de opção, lidar com formulários e assim por diante. Ele é focado mais ou menos exclusivamente em realizar as transações do Rack para nós, eliminando efetivamente o servidor da web. Capivara, portanto, tem que gerenciar a tradução de eventos iniciados pelo usuário em ações Rack::Test. Como exemplo, veja como o driver Capivara traduz o click evento em um elemento nas chamadas do método relevante para Rack::Test:

```

def clique
    if tag_name == 'a' method
        = self["data-method"] if driver.options[:respect_data_method] method ||= :get
        driver.follow(method, self[:href].to_s)

    elsif (tag_name == 'input' e %w(Enviar Imagem).include?(tipo)) ou ((tag_name == 'botão') e type.nil?
        ou type == "Enviar")
        Capivara::RackTest::Form.new(driver, form).submit(self)
    fim
fim

```

Capivara determina o tipo de elemento que está sendo clicado e então decide se deve dizer ao Rack::Test para seguir um link ou enviar um formulário.

Testando um aplicativo Sinatra

Tudo isso se torna mais tangível se olharmos para um exemplo mais realista. Usaremos o Sinatra simplesmente porque há menos código clichê do que o Rails, mas os princípios são exatamente os mesmos.

Considere o seguinte aplicativo Sinatra simples; ele nos permite concluir uma resenha de livro e devolver os dados enviados. Observe que não há persistência neste exemplo.

Arquivo de aplicação Sinatra – app.rb Este é o arquivo principal para nossa aplicação Sinatra que lida com a lógica do controlador:

```
requer 'sinatra'

class BookReview < Sinatra::Base

    obter '/form' fazer
        erb:formulario
    fim

    post '/submit' do @name
        = params[:name] @title =
        params[:title] @review =
        params[:review] @age = params[:age]
        erb :result

    fim

fim

BookReview.run! if __FILE__ == $0 #executar apenas se invocado a partir da linha de comando - caso contrário, vá para Capivara
```

Modelo de formulário – form.erb Este é um modelo ERB (formato de modelo HTML padrão do Ruby), que quando renderizado permitirá que um usuário envie uma resenha de livro:

```
<link rel="stylesheet" href="css/form.css"> <form action="/submit"
method="post"> <header id="header" class="info">

    <h2>Revisões de livros</h2>
    <div>Revise o último livro que você comprou...</div> </header>
```

Testando aplicativos Rails e Sinatra

```
<ul>
  <li>
    <rótulo para="nome">
      Seu nome
    </label>
    <input type="text" id="name" name="name" maxlegth="255" > </li>

    <label class="desc" for="idade">
      Faixa etária <
    label> <div>

      <select id="age" name="age"> <option value="-"
        selected="selected"> </option> <option value="<20" >
          Sub 20
        </option>
        <option value="20-50" >
          20 -50
        </option>
        <option value="50+" >
          Acima de 50
        </option> <
      select> </div> </li>

    <li>
      <label for="book_title">
        Título do livro
      </label>
      <input type="text" id="book_title" name="title" maxlegth="255" > </li>

    <li>
      <label for="avaliação">
        Sua revisão...
      </label>
      <textarea id="review" name="review" rows="10"
        cols="50"></textarea> </li> </li>

    <input type="submit" value="Enviar"/> </li> </ul> </form>
```

É assim que o formulário ficaria quando renderizado no navegador:

Book Reviews

Review the last book you purchased...

Your Name
Matt

Age Range
20 - 50

Book Title
Catch 22

Your Review...
One of the most ground-breaking books of the 21st Century!

Submit

Modelo de resultados – result.erb Este é outro modelo ERB, que renderizará a revisão salva:

```
<div class="saved_review"> <p>Você  
enviou o seguinte em: <%= Time.new.strftime("%Y-%m-%d %H:%M:%S") %> </p >  
  
<ul>  
<li>  
<p id="name">Nome: <%= @name %></p> </li>  
  
<li>  
<p id="age">Idade: <%= @age %></p> </li>  
  
<li>  
<p>Título do livro: <%= @title %></p> </li>
```

Testando aplicativos Rails e Sinatra

```
<li>
<p>Resenha do livro: <%= @review %></p> </li> <
ul> <a href="/form">Enviar outra resenha...</a> <
div>
```

É assim que a avaliação enviada aparecerá no navegador:

You submitted the following on: 2013-04-12 19:12:22

- Name: Matt
- Age: 20-50
- Book Title: Catch 22
- Book Review: One of the most ground-breaking books of the 21st Century!

[Submit another review...](#)

Quando o aplicativo é executado a partir da linha de comando usando ruby app.rb, você pode navegar para <http://localhost:4567/form> e enviar uma resenha do livro; isso ecoará os detalhes enviados em <http://localhost:4567/result>.

Portanto, agora que temos um aplicativo Sinatra funcionando, precisamos ver como testá-lo usando o Capivara e mostrar crucialmente a diferença entre testar com Rack::Test como driver em oposição ao Selenium WebDriver.

Testando com Rack::Test Do ponto de vista da

Capivara, você interage com seu aplicativo Sinatra/Rails exatamente da mesma forma que faria com um aplicativo remoto. É apenas a configuração que difere um pouco.

Por exemplo, aqui estão algumas definições de etapas do Cucumber com sua implementação Capivara para automatizar o preenchimento do formulário de resenha do livro e verificar os resultados enviados:

```
Dado(/^estou em um site de resenhas de livros$/) faça
    visite('/formulario')
fim

Quando(/^eu enviar uma resenha de livro$/)
    preencha_in 'nome', :com => 'Matt'
```

```
fill_in 'title', :with => 'Catch 22' fill_in 'review', :with =>
'Tudo bem, eu acho...' selecione '20 - 50', :from => 'idade' click_on 'Enviar'
```

fim

```
Then(/^Devo ver os detalhes salvos confirmados$/) faça
page.should have_text 'Você enviou o seguinte em:' find('#name').should have_text
'Matt' find('#age').should have_text '20-50' find('#review').should have_text 'Todo
bem, eu acho...' find('#title').should have_text 'Catch 22'
```

fim

Essa implementação não é novidade para nós até agora e, felizmente, isso não mudará, independentemente do driver que escolhermos, essa é a beleza do Capivara.

No entanto, a configuração é algo que precisamos examinar com um pouco mais de detalhes. Aqui está um exemplo de arquivo env.rb que você pode ter em seu projeto Cucumber para testar seu aplicativo Rails ou Sinatra:

```
require 'capivara/pepino' require 'rspec/
expectations' require_relative '../sinatra/
app'

Capivara.default_driver = :rack_test

Capivara.register_driver :selenium do |app|
  Capivara::Selenium::Driver.new(app, :browser => :chrome) end
```

Capybara.app = BookReview

Aqui apresentamos apenas um novo conceito, que é definir o valor de Capybara.app para a classe principal de nosso aplicativo Sinatra, que neste caso é a classe BookReview .

O Capybara.default_driver também está definido como :rack_test, então quando rodarmos os testes eles usarão Rack::Test ao invés de Selenium WebDriver. Nenhum navegador será aberto e nenhuma solicitação HTTP será feita, o que significa que eles serão executados muito rapidamente; ainda assim, como discutimos antes, eles ainda exercitam toda a pilha de aplicativos.

No entanto, ainda queremos a opção de executar esses testes usando o Selenium em um navegador real. Para fazer isso, simplesmente alteraremos o driver padrão:

```
Capivara.default_driver = :selenium
```

Testando aplicativos Rails e Sinatra

Se você executar seus testes novamente, notará que como num passe de mágica sua aplicação está rodando no servidor e as páginas estão disponíveis para os testes acessarem. Se você olhar de perto, notará que o aplicativo está sendo executado em uma porta aparentemente aleatória. Como o aplicativo é um aplicativo Rack, o Capivara é capaz de iniciar o servidor de aplicativos para você e pará-lo ao final dos testes. Nesse caso, é claro que você está exercitando seu aplicativo em HTTP e em um navegador real, portanto, os testes serão executados significativamente mais lentos.

Apenas para destacar os problemas de velocidade, aqui estão duas execuções do mesmo cenário: a primeira usando Rack::Test e a segunda usando Selenium WebDriver. Como você pode ver, a diferença de velocidade é significativa:

```
$ bin/cucumber -r features features/chapter3/sinatra.feature
```

Recurso: Usando Capivara e Rack-Test para interagir com o aplicativo Sinatra

Cenário: Resenha completa do livro sinatra.feature:3	# características/capítulo3/
<i>Dado que estou em um site de resenhas de livros</i> passos/sinatra.rb:1	# características/capítulo3/
<i>Quando envio uma resenha de livro</i> passos/sinatra.rb:5	# características/capítulo3/
<i>Então devo ver os detalhes salvos confirmados # features/chapter3/ steps/sinatra.rb:13</i>	

1 cenário (1 passou)

3 passos (3 passados)

0m0.064s

```
$ bin/cucumber -r features features/chapter3/sinatra.feature
```

Recurso: Usando Capivara e Selenium-Webdriver para interagir com Sinatra

Aplicativo

Cenário: Resenha completa do livro sinatra.feature:3	# características/capítulo3/
<i>Dado que estou em um site de resenhas de livros</i> passos/sinatra.rb:1	# características/capítulo3/
<i>Quando envio uma resenha de livro steps/</i> sinatra.rb:5	# características/capítulo3/

Então devo ver os detalhes salvos confirmados # features/chapter3/ steps/sinatra.rb:13

1 cenário (1 passou)

3 passos (3 passados)

0m4.079s



Como o Capybara inicia seu aplicativo, isso significa que você não precisa definir o host base usando Capybara.app_host ou usar URLs completos em suas chamadas para o método visit. Você pode simplesmente usar o caminho relativo para a página, como visitar '/form'.

Qual driver usar e quando?

É importante entender que mesmo quando temos Rack::Test disponível, nem sempre é a melhor opção. Como mencionado anteriormente, usar o driver Rack::Test testará a pilha completa de seu aplicativo e executará testes mais rapidamente do que usar qualquer solução baseada em navegador sem cabeça ou de outra forma.

No entanto, há duas considerações importantes:

- Rack::Test não é um navegador real
- Rack::Test não executa nenhum código do lado do cliente

É importante ter em mente que existem recursos de um servidor web e navegador que afetarão o comportamento do aplicativo em teste que não testaremos ao usar o Rack::Test. Por exemplo, os cabeçalhos de cache HTTP não estão em jogo, então ainda há um caso para teste neste contexto.



Rack::Test mantém um JAR de cookie, portanto, o comportamento dos cookies ainda deve ser replicado como em um navegador real.

Finalmente, e talvez o mais importante, o Rack::Test não testará nenhum JavaScript do lado do cliente do seu aplicativo, pois lida apenas com o código executado no servidor.

Se você tiver um aplicativo Rails ou Sinatra totalmente dependente de JavaScript e ativos estáticos para fornecer funcionalidade principal, será necessário garantir que isso seja coberto por seus testes Capivara que usam o Selenium WebDriver.

Como regra geral, seria aconselhável aproveitar o poder e o desempenho do Rack::Test para executar testes de pilha completa que validam o núcleo de sua integração do lado do servidor. Além disso, sempre certifique-se de ter pelo menos um conjunto de testes de sanidade ou fumaça executados usando o Selenium WebDriver em um navegador real.

Testando aplicativos Rails e Sinatra

A Capivara te ajuda aqui; se você estiver usando o Cucumber, basta marcar os cenários que exigem Suporte a JavaScript com @javascript e em RSpec coloque :js => true em suas especificações como no exemplo a seguir:

```
#Para Pepino
@javascript
Cenário: Resenha completa do livro

#Para RSpec,
descreva "Revisão do livro"
it "permite que o usuário envie uma avaliação", :js => true do
```

A Capivara então trocará os drivers conforme necessário. Se você não quiser usar o Selenium WebDriver como seu driver JavaScript, pode alterá-lo para outro driver, como Capybara-WebKit, usando Capybara.javascript_driver = :webkit.

Uma nota sobre Rails/RSpec e Capivara

Se você usa Active Record em sua aplicação rails para gerenciar transações de banco de dados, você pode ou não estar ciente do conceito de **Fixtures Transacionais**.

O princípio aqui é que, ao usar estruturas de teste compatíveis, o banco de dados será limpo entre cada caso de teste, garantindo que não haja poluição entre os testes.

Este código faz parte de ActiveRecord::TestFixtures e pode ser ativado ou desativado na estrutura de teste que você está usando.

Os acessórios transacionais funcionarão bem quando sua estrutura de teste estiver sendo executada no mesmo processo que seu aplicativo, como quando você estiver usando Rack::Test com Capybara. No entanto, assim que você executa sua aplicação em um servidor web e usa um driver como Selenium WebDriver ou Capybara-WebKit, seus testes começam a rodar em um processo diferente e, portanto, não tem visibilidade do código rodando no servidor. Portanto, as transações não serão revertidas após cada teste.

Nesse caso, o que a maioria das pessoas faz é usar o Database Cleaner (https://github.com/bmabey/database_cleaner) para garantir que o banco de dados seja truncado entre cada teste. O README do projeto lhe dará muita ajuda na configuração, caso você escolha esta opção.

Resumo

O Capybara oferece suporte a testes de Rails, Sinatra e todos os outros aplicativos Rack prontos para uso e esse suporte é uma parte central da biblioteca.

Cobrimos o que "compatível com Rack" realmente significa e isso é crucial para entender por que o Capybara em conjunto com o Rack::Test é tão adequado para testar aplicativos usando Rails ou Sinatra.

Por fim, consideraremos qual driver usar e quando, destacando que às vezes você ainda precisará usar o Selenium WebDriver para testar algumas das funcionalidades do seu aplicativo. Claramente, no que diz respeito ao JavaScript, Rack::Test será inútil e você terá que usar o driver Selenium.

Lembre-se do mantra "fail fast" e use Rack::Test para executar testes full-stack rápidos antes de usar as armas pesadas do Selenium.

Se o seu aplicativo é JavaScript pesado, não se preocupe, veremos como o Capivara lida com isso com facilidade no próximo capítulo.

Machine Translated by Google

4

Lidando com Ajax, JavaScript e Flash

Já faz muito tempo desde que os aplicativos da Web consistiam principalmente em HTML estático. A maioria dos aplicativos da Web modernos possui grandes quantidades de JavaScript que modificam o DOM no cliente (no navegador). A automatização desses aplicativos requer um nível adicional de conscientização por parte do desenvolvedor que cria os testes. O JavaScript no DOM não é síncrono; manipuladores de eventos são usados para alterar e modificar a página conforme ela é carregada e os usuários interagem com ela e, como tal, nossos testes precisam ser robustos o suficiente para lidar com esse comportamento assíncrono.

Felizmente, o Capivara foi construído com esse princípio em sua essência, o que torna as coisas muito fáceis para nós. Neste capítulo, trabalharemos com exemplos que demonstram isso.

Também consideraremos elementos da página que à primeira vista parecem impossíveis de automatizar, como componentes flash ou elementos HTML5, por exemplo, a tag canvas .

Ajax e JavaScript assíncrono

Se você não estiver muito familiarizado com o JavaScript ou sua função no aplicativo da web, uma breve visão geral pode valer a pena.

Os interpretadores JavaScript existem em todos os navegadores da Web modernos e permitem que os desenvolvedores adicionem funcionalidade do lado do cliente às suas páginas, alterando o conteúdo dinamicamente assim que a página é carregada.

Lidando com Ajax, JavaScript e Flash

A API exposta via JavaScript é amplamente baseada em eventos, portanto, o código JavaScript em execução na página pode se registrar para ouvir um evento específico e fornecer uma função que será chamada quando esse evento ocorrer. Um exemplo disso pode ser uma entrada de pesquisa com um recurso de sugestão automática, em que o JavaScript escutará eventos de pressionamento de tecla e começará a atualizar a página com sugestões de pesquisa com base na entrada de texto do usuário.

Ajax (Asynchronous JavaScript and XML) é apenas um subconjunto da API disponível via JavaScript e está relacionado à capacidade de carregar conteúdo na rede de forma assíncrona usando XML ou JSON (JavaScript Object Notation).

Por que isso é problemático para a automação?

O carregamento básico da página é detectado pela maioria das ferramentas de automação prontas para uso. Por exemplo, o Selenium sabe quando o DOM está pronto e não permite a interação com a página até que isso ocorra. No entanto, o Selenium não pode saber como o JavaScript do seu aplicativo específico modificará o DOM e, portanto, essa responsabilidade recai sobre você como desenvolvedor do teste.

Felizmente, o Capivara construiu um muro de defesa contra esse problema que remove o máximo possível da fragilidade de nossos testes.

Capivara e JavaScript assíncrono

Vejamos precisamente como o Capivara nos ajuda a gerenciar o JavaScript assíncrono em nossos aplicativos.

Os trechos de código a seguir mostram primeiro algumas marcações simples de uma página da Web e, em seguida, algum JavaScript que carrega imagens do Flickr usando Ajax e as anexa como tags de imagem à marcação existente.

Aqui, a parte do corpo do documento mostra o estado inicial do DOM antes do botão ser clicado e o JavaScript ser executado:

```
<corpo>
  <div id="principal">
    <input type="button" id="load" value="Carregar imagens" /> <div
      class="section">
        <div id="images"></div> </div>
    </div> </body>
```

E aqui está o JavaScript que carrega as imagens e as anexa ao documento:

```
<script> $  
  (documento).ready(function() { $  
    ('#load').click(function(){  
      var flickerAPI = "http://api.flickr.com/services/feeds/photos_public.gne?  
      jsoncallback=?"; $.getJSON( flickerAPI, { tags: "capybara", tagmode: "any", format:  
      "json" } ) .done(function( data ) $.each( data.items, function( i, item )  
  
      {  
        {  
          $( "" ).attr( "origem",  
          item.media.m ) .appendTo( "#imagens" ); se (i === 3)  
  
          { retorna falso;  
        }  
  
      }  
  
    } ); } );  
</script>
```

A página renderizada se pareceria com a captura de tela a seguir:



The screenshot shows a browser window with three images of capybaras. The first image is a close-up of a capybara's head in grass. The second image shows a capybara in a cage with a person standing nearby. The third image is a close-up of a capybara's body. Below the images is a developer tools network tab showing six requests made to Flickr's API to load the images.

URL	Status	Domain	Size	Remote IP	Timeline
▶ GET jquery.min.js	304 Not Modified	ajax.googleapis.com	32 KB	74.125.132.95:80	9ms
▶ GET photos_public.gne?...on&_=1363	200 OK	api.flickr.com	1.8 KB	68.142.214.24:80	
▶ GET 8577606165_629b425154_m.jpg	304 Not Modified	farm9.staticflickr.com	28.7 KB	77.238.160.184:80	
▶ GET 8577665924_2d83eba7b5_m.jpg	304 Not Modified	farm9.staticflickr.com	19 KB	77.238.160.184:80	
▶ GET 8576457661_10c2d79675_m.jpg	304 Not Modified	farm9.staticflickr.com	23.3 KB	77.238.160.184:80	
▶ GET 8577556028_0cdbbd26d2_m.jpg	304 Not Modified	farm9.staticflickr.com	19.7 KB	77.238.160.184:80	
6 requests			124.6 KB	(122.8 KB from cache)	

Lidando com Ajax, JavaScript e Flash

Neste exemplo, a solicitação Ajax é feita usando a biblioteca jQuery (<http://jquery.com/>) e foi feita usando JSONP (JSON with Padding), que nos permite fazer uma solicitação de dados entre domínios.

Quando um usuário clica no botão **Carregar imagens**, isso faz com que quatro solicitações de rede sejam feitas à API do Flickr para solicitar imagens marcadas com capivara. Você pode ver essas solicitações no console do Firebug (<http://getfirebug.com>), conforme mostrado na captura de tela anterior. Isso é completamente assíncrono, portanto, quando estamos testando "de fora para dentro" usando Capivara e Selenium, não há como saber quando essas solicitações serão concluídas.

Vamos supor que já concluímos uma etapa que nos diz para clicar no botão **Carregar imagens** e agora queremos escrever um teste para verificar se as quatro imagens foram carregadas e adicionadas à página.

```
Then(/^Eu devo ver todas as imagens carregadas com sucesso$/) do find(:xpath, '//img[4]')
```

fim



A expressão xpath anterior verifica os quatro elementos img.
Isso também é possível com CSS3; para fazer isso, inclua o seguinte código:
`find('div > img:first-child:nth-last-child(5)')`

Incrivelmente, isso é tudo que você precisa fazer. Você pode supor que isso não funcionaria porque nosso teste simplesmente carregará a página e verificará quatro elementos de imagem que provavelmente não estariam lá, pois o navegador ainda os está carregando.

Felizmente, o Capybara incorpora a lógica de repetição em grande parte de sua API, para que não precisemos nos preocupar em fazer isso sozinhos.

A quantidade de tempo que o Capivara tentará novamente é padronizada para dois segundos, mas pode ser substituída definindo o atributo `default_wait_time`; se você estiver usando o Cucumber, você pode adicionar isso ao seu arquivo `env.rb`:

```
exigem 'empacotamento/
configuração' exigem 'capivara/pepino'
exigem 'rspec/expectativas'
```

```
Capybara.default_driver = :selenium
Capybara.default_wait_time = 10
```

É importante observar que essas novas tentativas assíncronas se aplicam apenas onde o driver que você está usando oferece suporte a JavaScript. Por exemplo, ao usar o Selenium, eles certamente funcionarão. Se o driver não suportar JavaScript, não há necessidade de o Capivara aguardar, pois basta aguardar o carregamento da página.

Métodos que lidam com assíncrono JavaScript

Tendo visto como a lógica de repetição do Capivara funciona ao usar uma busca simples, agora você precisa ver até que ponto isso se estende pela API.

Tanto os localizadores quanto os correspondentes de Capivara esperam por JavaScript assíncrono.

localizadores

Você encontrou os métodos localizadores anteriormente neste livro e todos esses métodos têm a funcionalidade de espera integrada.

Conforme mostrado no exemplo anterior, `find(:xpath, '//img[4]')` esperou tempo suficiente para permitir que o JavaScript assíncrono fosse executado e que o navegador carregasse as imagens. Os outros localizadores que têm essa capacidade são:

- `find_field`
- `encontrar_link`
- `botão_encontrar`
- `encontrar_por_id`
- `todos`
- `primeiro`

Matchers

Os correspondentes usados para validar o conteúdo na página também aguardarão o JavaScript assíncrono; por exemplo, para usar um RSpec Matcher, você pode alterar o exemplo anterior da seguinte forma:

```
Then(/^Devo ver todas as imagens carregadas com sucesso$/) faça
  page.should have_selector(:xpath, '//img[4]') end
```

Lidando com Ajax, JavaScript e Flash

Assim como nos métodos localizadores, você obtém esse comportamento gratuitamente com todos os correspondentes e também com suas contrapartes negativas, como segue:

- has_xpath? / has_no_xpath? •
- has_css? / has_no_css? •
- has_content? / has_no_content? (Semelhante a has_text? / has_no_text?) •
- has_link? / has_no_link? • has_button? / has_no_button? • has_field? / has_no_field? • has_checked_field? / has_no_checked_field? •
- has_unchecked_field? / has_no_unchecked_field? • has_select? / has_no_select?
- has_table? / has_no_table?

Pegadinhas

Apesar dos poderosos mecanismos de defesa assíncronos da Capivara, ainda é possível ser pego. Vejamos outro exemplo:

```
<html>
  <cabeça>
    <title>Exemplos de assíncrono </
    title> <script src="http://
    ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"> </script>
    <script> $( document ).ready(function() { var addText = function() {

      $('.section').attr('estilo', 'visibilidade:visível;');

      } setTimeout(addText, 5000); });
    </script> </head> <body> <div id="main">

      <div class="section" style="visibility:hidden;">
        Rochas da Capivara
      </div> </div> </body> <
    html>
```

Capítulo 4

Esta página tem algum código JavaScript que é executado pelo menos cinco segundos após o carregamento da página e define o estilo de visibilidade no div com seção de classe como visibilidade:visível. Isso torna o texto **Capivara Rocks** visível para o usuário.

Você pode pensar que o código a seguir funcionaria e esperaria tempo suficiente para verificar se o elemento se torna visível:

```
Então(/^Capivara espera o elemento ficar visível$/) faça
    find('.section').visible?.should == true
fim
```

Neste exemplo, Capivara não vai esperar que o elemento fique visível. A razão é que Capivara simplesmente espera que um elemento com a classe correta exista dentro do DOM, então imediatamente verifica a visibilidade do elemento e retorna.

Isso ocorre porque o visible? method não é um dos localizadores ou matchers da Capivara; é simplesmente um método usado para consultar o estado de um elemento a qualquer momento.

Você pode obter esse comportamento, se desejar, escrevendo seu próprio método wait_for personalizado :

```
requer 'tempo'

def wait_for(wait = 8)
    timeout = Time.new + espera

    while (Tempo.novo < tempo limite)
        retorno se (rendimento)
    fim
    raise "Condição não atendida em #{wait} segundos"
fim

Quando(/^eu visito uma página que torna um elemento visível com um atraso$/) faça
    visite 'http://localhost/html/asynch.html'
fim

Then(/^Capivara espera que o elemento fique visível$/) do wait_for(10) { find('.section').visible? }

fim
```

O método wait_for aceita um tempo de espera em segundos como argumento e então executa qualquer bloco que é passado para ele. Em seguida, ele executa continuamente o bloco, até que retorne verdadeiro ou o valor do tempo limite seja excedido.

Lidando com Ajax, JavaScript e Flash

Se você se encontrar nessa situação, provavelmente vale a pena considerar se você pode usar um método ou seletor que aproveite a funcionalidade de espera integrada do Capivara. Por exemplo, você pode passar um parâmetro adicional para o `find` método para garantir que ele espere o elemento ficar visível:

```
Então(/^Capivara espera o elemento ficar visível$/) faça
    find('.section', :visible => true)
fim
```

A API do Capivara é tão elegante que usá-la quase sempre será uma opção melhor do que o código que você pode tentar escrever para lidar com esses problemas.

Flash e HTML5 – elementos de caixa preta Assim como o uso de JavaScript assíncrono e Ajax ultrapassou os limites dos aplicativos da web, existem outros componentes fora da marcação estática que tornam o teste de aplicativos da web modernos um desafio.

Exemplos de tais componentes incluem:

- Aplicativos Flash, como jogos, players de vídeo e assim por diante
- Tag de tela HTML5 usada para desenhar usando uma API JavaScript
- Tag de vídeo / áudio HTML5

Todos esses componentes têm algo em comum, ou seja, possuem funcionalidades que operam até certo ponto fora do contexto do DOM. Não podemos inspecionar seus componentes internos usando as técnicas que discutimos até agora, porque todas essas técnicas são baseadas na inspeção do DOM em busca de elementos específicos, seus atributos e valores de texto.

Por exemplo, seu aplicativo pode desenhar dinamicamente diagramas de gráfico de pizza em uma tela com base em alguma entrada do usuário e você, é claro, gostaria de testar isso.

No entanto, quando você inspeciona o DOM, tudo o que você vê é o seguinte código:

```
<canvas id="org_chart" width="500" height="500"></canvas>
```

Isso ocorre porque não há nada que você possa inspecionar, pois o navegador está renderizando um bitmap dinamicamente com base nas instruções passadas por meio da API JavaScript.

Você deseja testar esses componentes usando o Capivara, mas o problema aqui não é o Capivara; você precisa expor uma API testável desses componentes para permitir que qualquer ferramenta de automação de navegador tenha uma chance de agregar valor.

Clarão

Os aplicativos Flash não são diferentes de alguns dos elementos HTML5. São caixas pretas contendo código compilado e embutido na página. A única maneira de tornar um aplicativo Flash em sua página acessível para ferramentas de teste é garantir que você e sua equipe de desenvolvimento criem uma API testável que seja exposta via JavaScript.

O Flash permite isso oferecendo a API de interface externa (http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/external/ExternalInterface.html), por meio da qual uma função ActionScript pode ser disponibilizada para JavaScript. Além disso, você pode usar FlashVars (<http://helpx.adobe.com/flash/kb/pass-variables-swfs-flashvars.html>) para passar dados para o objeto incorporado de JavaScript ou por meio de atributos em elementos.

De uma perspectiva de automação de teste, há pouca diferença em princípio entre um componente Flash e alguns dos componentes HTML5 de caixa preta. Nos exemplos a seguir, o HTML5 é usado em vez do Flash, para que você não precise se preocupar com a compilação do ActionScript ou com a criação de arquivos SWF, o que está fora do escopo deste livro.

Expondo uma API testável

O elemento de áudio HTML5 é outro exemplo de componente de caixa preta, semelhante a um reprodutor de vídeo Flash.

Imagine que sua equipe de desenvolvimento deseja incorporar um reprodutor de áudio nas páginas do site, escolhendo o elemento de áudio HTML5 em torno do qual construir o reprodutor.

Inicialmente, eles apenas criam alguns controles personalizados; posteriormente, eles pretendem adicionar recursos mais avançados, como listas de reprodução definidas pelo usuário e links para sites de artistas da faixa atual.

O pico inicial da equipe apenas apresenta os seguintes controles personalizados:

- A pausa
- Procurar

Como membro da equipe responsável pela automação de teste, você fica coçando a cabeça; como você pode testar se clicar em reproduzir realmente reproduz a música e a pausa a interrompe? Embora os controles nativos sejam visíveis, você não pode interagir diretamente com eles, pois eles fazem parte do Browser Object Model (BOM) e não do DOM.

Lidando com Ajax, JavaScript e Flash

A página a seguir demonstra como uma implementação muito ingênua disso pode parecer:

```
<!DOCTYPE html>
<html>
<head>
<title>Exemplos de HTML5</title> <script
src="http://ajax.googleapis.com/ajax/libs/jquery/1.9.1/jquery.min.js"> </script>
<script> $( documento).pronto(funcção() {

    $('#play').click(function(){ $('audio')[0].play(); });

    $('#pause').click(function(){ $('audio')
        [0].pause(); });

    $('#seek').click(function(){
        var val = $('#seekval').val(); $('audio')
        [0].currentTime = val; });

});

</script> </
head> <body>

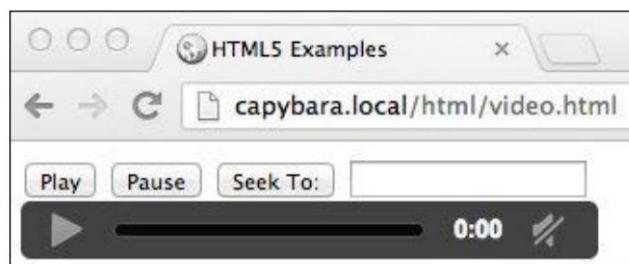
<div id="principal">
    <input type="button" id="play" value="Play" /> <input type="button"
    id="pause" value="Pause" /> <input type="button" id="seek" value="Seek
    To:" /> <input id="seekval" value="" /> <div class="section">
        <audio src='http://www.vorbis.com/music/Hydrate-
        Kenny_Beltrey.ogg' controls="true"> </audio> </div>
    </div> </
body> <
html>
```

Este exemplo grava um elemento de áudio HTML5 na página, com um elemento src apontando para o arquivo que queremos reproduzir.

O exemplo anterior usa jQuery (daí o uso de \$) para ajudar a localizar elementos e anexar eventos. Isso foi planejado para ser breve e também porque muitas das equipes com as quais você trabalha usarão jQuery. Claro que não há dependência aqui; você poderia facilmente ter usado funções como getElementById e getElementsByTagName.

Existem botões para **Reproduzir**, **Pausar** e **Buscar**, que usam funções de retorno de chamada para conectar-se à API JavaScript do elemento de áudio para reproduzir, pausar ou procurar na música.

Esta página pode se parecer com a captura de tela a seguir quando renderizada no navegador:



Vale a pena notar novamente que você não pode interagir com os controles nativos (pelo menos não sem algumas técnicas seriamente desagradáveis), então como você valida o comportamento de seus botões e controles personalizados?

Nem toda a esperança está perdida e, de fato, acaba sendo bastante simples. Observar atentamente a especificação de áudio do HTML5 revela que há eventos que são gerados pelo navegador quando um comportamento específico é invocado. Podemos explorar esses eventos e, de repente, esse comportamento torna-se imediatamente testável.

Páginas de teste – eis o poder!

Se você vem de um histórico de teste, normalmente tende a sempre querer testar o aplicativo da mesma forma que um usuário o usaria e pode não se sentir confortável em colocar um componente em uma página específica de teste em vez de testá-lo "in situ".

Posso simpatizar com esse pensamento; mas muitas vezes faz sentido retirar um componente, especialmente quando é uma parte isolada da página, como no exemplo do nosso reprodutor de áudio. Você ainda desejará escrever uma fina camada de testes com ela incorporada em uma página real. Mas, para realmente colocá-lo à prova, ajudará a isolá-lo. A razão para fazer isso é que podemos explorar todos os eventos JavaScript, gerar algumas informações de depuração na página e, em seguida, usar o Capybara para coletar essas informações e validar se estão corretas.

Lidando com Ajax, JavaScript e Flash

Vamos atualizar nosso exemplo com algumas informações coletadas de eventos gerados e, em seguida, enviá-los para a página.

Elementos adicionais são adicionados à marcação, que conterá o estado atual do player e a posição da música:

```
<corpo>
  <div id="principal">
    <input type="button" id="play" value="Play" /> <input type="button"
    id="pause" value="Pause" /> <input type="button" id="seek"
    value="Seek To:" /> <input id="seekval" value="" />

  <div class="seção">
    <audio src="http://www.vorbis.com/music/ Hydrate-
      Kenny_Beltrey.ogg' controls="false"> </audio> </div>
    <div>Estado do jogador: <span id="log" >parado</div>
    <div>Posição da música: <span id="time">0.00</div> </
    body>
```

O código JavaScript precisa então ser aumentado para preencher esses novos elementos:

```
<script> $
(documento).ready(function() {

  var estadodeatualização = function(estado) {
    $('#log').text(estado); };

  var updateTime = function() {
    var currTime = $('audio')[0].currentTime; var currTime =
    Math.round(currTime*100)/100 $('#time').text(currTime); }

  $('#play').click(function(){ $('audio')
    [0].play(); });

  $('#pause').click(function(){ $('audio')
    [0].pause(); });

});
```

```
});

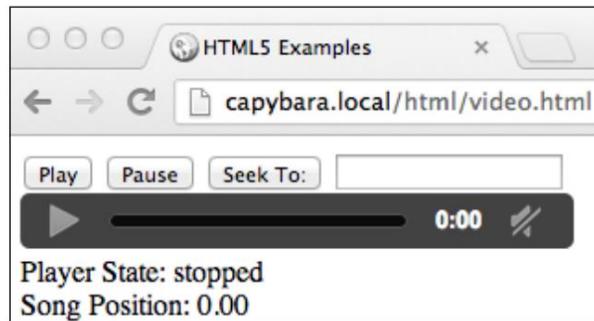
$('#seek').click(function(){
    var val = $('#seekval').val();
    $('audio')[0].currentTime = val; });

$('audio')[0].addEventListener('reproduzindo', function() { updateState('reproduzindo'); }, false);
$('audio')[0].addEventListener('pause', function() { updateState('paused'); }, false); $
('audio')[0].addEventListener('timeupdate', function() { updateTime(); }, false); });


```

Agora que as três chamadas de função addEventListener foram adicionadas ao elemento de áudio , essas informações podem ser enviadas para a página usando os elementos com os IDs log e time.

Agora sua página contém informações de depuração que você pode usar para testar o comportamento dos controles personalizados usando Capivara:



Como investimos o esforço para desenvolver uma página testável, a automação real usando Capivara/ Selenium é direta. Uma lição importante aqui é que, embora haja uma compensação ao testar o componente em uma página dedicada, isso torna nossa automação de teste muito menos frágil, pois o teste simplesmente precisa se preocupar em clicar nos botões e raspar o texto, simples !

Nosso recurso e etapas podem ser algo como o seguinte:

Funcionalidade: validar controles personalizados no reprodutor de áudio HTML5

Cenário: validar o controle 'play'

Lidando com Ajax, JavaScript e Flash

Dado que eu visito uma página com um reprodutor de áudio HTML5 personalizado
Quando eu clico para tocar uma música
Então a música toca

Dado(/^eu visito uma página com um reprodutor de áudio HTML5 personalizado\$/) do
visite 'http://localhost/html/html5.html'
fim

Quando(/^eu clicar para tocar uma música\$/)
clique_em 'Play'
fim

Então(/^a música toca\$/) faça
find('#log').should have_text 'playing'
fim

Deve ser óbvio agora que você pode facilmente adicionar testes para cenários de pausa e busca. Ao fazer isso, você terá tornado um componente anteriormente não testável perfeitamente testável.

Componentes de teste "in situ"

Ainda pode haver ocasiões em que testar um componente isoladamente em uma página de teste dedicada não é apropriado e todos os testes precisam ser feitos na página que implementará o componente. No exemplo do componente reprodutor de áudio, é improvável que você tenha a opção de enviar as informações do evento para as páginas de seus aplicativos reais.

Felizmente, ainda temos opções para expor os eventos que tornam o componente testável, embora isso signifique que seu código de teste não será tão limpo ou sustentável; essa é a compensação.

Ao usar o Capybara com Selenium (e outros drivers suportados, como o Capybara-WebKit), você tem a opção de executar JavaScript arbitrário na página atual. Por exemplo, a seguinte linha de código obteria todos os elementos div na página:

```
page.evaluate_script('document.getElementsByTagName("div")')
```

Capítulo 4

Como uma possível solução para seu problema com a exposição de informações de depuração em uma página de aplicativo real, você pode concordar com a equipe em alterar o código do aplicativo e criar objetos JavaScript na página com essas informações anexadas. Para fazer isso para o exemplo de áudio HTML5, o JavaScript precisaria apenas ser alterado da seguinte forma:

```
<script>

var audioDebug = {
    estado: 'parado',
    tempo: 0,00
};

$(documento).pronto(function() {

    var updateState = function(estado) { audioDebug.state
        = estado; };

    var updateTime = function() {
        var currTime = $('audio')[0].currentTime; var currTime =
        Math.round(currTime*100)/100; audioDebug.time = currTime; };

    $('#play').click(function(){ $('audio')
        [0].play(); });

    $('#pause').click(function(){ $('audio')
        [0].pause(); });

    $('#seek').click(function(){
        var val = $('#seekval').val(); $('audio')
        [0].currentTime = val; });

    $('audio')[0].addEventListener('reproduzindo', function() { updateState('reproduzindo'); },
    false); $('audio')[0].addEventListener('pause', function() { updateState('pausado'); },
    false); $('audio')[0].addEventListener('timeupdate', function() { updateTime(); },
    false);

});;
</script>
```

Lidando com Ajax, JavaScript e Flash

No código anterior, o objeto audioDebug é criado para conter as informações de depuração quando os eventos são disparados. Como poluir o namespace global em JavaScript é desencorajado, você desejará criar esse objeto dentro do namespace do código JavaScript do aplicativo.

As etapas do Cucumber agora podem ser alteradas para refletir esta nova estratégia:

```
Dado(/^eu visito uma página com um reprodutor de áudio HTML5 personalizado$/) do
    visite 'http://localhost/html/html5.html'
fim

Quando(/^eu clicar para tocar uma música$/)
    clique_em 'Play'
fim

Then(/^the song plays$/) do wait_for do
    page.evaluate_script('audioDebug.state').should == 'playing' end
fim
```

A única diferença é que agora você executa algum JavaScript usando o método `avalia_script` do Capybara que obtém a propriedade `state` do objeto `audioDebug`.



Você terá que empregar a função customizada `wait_for` que foi discutida anteriormente. Como o Capivara não cria nenhum tipo de espera ao executar JavaScript arbitrário, é provável que, se você clicar em **Reproduzir** e verificar imediatamente o estado do objeto `audioDebug`, ele não tenha tido tempo de atualizar seu estado.

Ao discutir essas estratégias com sua equipe, você pode descobrir que os desenvolvedores podem pensar em algumas abstrações inteligentes para gerenciar eventos JavaScript no aplicativo. Por exemplo, você pode implementar um padrão que envia os eventos para um objeto proxy e, em seguida, ter diferentes objetos de estratégia que podem ser ativados/desativados e determinar o que é feito com os dados; por exemplo, se os dados são gravados em um objeto de depuração ou no console para criação de log. Nem é preciso dizer que você desejará desativar essa saída de depuração quando o aplicativo for implantado na produção.

Resumo

Lidar com JavaScript assíncrono inevitavelmente será um desafio que você terá de enfrentar. Vimos como o Capybara facilita isso implementando esperas integradas em todos os métodos que envolvem encontrar algo na página ou interagir com elementos que podem não estar imediatamente visíveis.

Além disso, aceitamos o desafio de testar os componentes da caixa preta e usamos um reprodutor de áudio HTML5 como exemplo. Também vimos que a abordagem mais comum para resolver esse problema é expor uma API JavaScript testável e, se possível, implementar páginas específicas de teste com a saída de depuração exibida na página pronta para validar usando Capivara.

Machine Translated by Google

5

Tópicos Ninja

No capítulo final deste livro, parece apropriado olharmos além da API básica e da funcionalidade que o Capybara oferece. Agora você tem todas as habilidades necessárias para automatizar sua aplicação usando o Capivara, seja ela uma aplicação Rails/Sinatra ou uma aplicação web escrita em qualquer outro framework.

Este capítulo irá garantir que você se sinta confortável usando Capivara fora do Pepino. Ele também mostrará como você pode acessar a funcionalidade em seu driver escolhido que não é mapeado pela API do Capivara e apresentará alguns dos outros drivers que você pode não ter encontrado.

Especificamente, abordaremos:

- Usando o Capivara fora do Cucumber •
Interações avançadas e acesso direto ao driver • Configuração
avançada do driver • Ecossistema do driver - algumas opções
alternativas

Usando capivara fora do pepino

Até agora, a maioria dos exemplos neste livro foram definidos no contexto das definições de etapas do Pepino, já que esta é de longe a maneira mais comum pela qual as pessoas usam a Capivara.

No entanto, o Capybara não está de forma alguma acoplado ao Cucumber e pode ser usado em qualquer configuração que você desejar, dentro de Test::Unit, RSpec ou apenas do código Ruby vanilla. Na verdade, se você estiver usando o Cucumber, mas quiser abstrair alguma lógica de suas definições de etapa e em Objetos de página, ainda precisará considerar como usar o Capivara fora do mundo do Cucumber (<https://github.com/cucumber/pepino/wiki/A-Whole-New-World>).

Tópicos Ninja



O padrão Page Object é uma maneira simples de estruturar sua estrutura de teste se você estiver lidando com aplicativos baseados em navegador. Você simplesmente modela cada página (ou cada componente principal da interface do usuário) como um objeto. Aspectos como seletores CSS podem ser armazenados como propriedades e você pode implementar métodos para manipular a página. *Simon Stewart*, um dos principais desenvolvedores do projeto Selenium WebDriver, fornece algumas práticas recomendadas úteis se você optar por implementar esse padrão, que estão disponíveis em <https://code.google.com/p/selenium/wiki/PageObjects>.

Incluindo os módulos

A primeira opção que você tem para usar a Capivara fora do Pepino é incluir o Módulos DSL em seus próprios módulos ou aulas:

```
require 'capivara/dsl' require 'rspec/
expectations'

Capivara.default_driver = :selenium

módulo MyModule include
  Capivara::DSL include
  RSpec::Matchers

  def play_song
    visite 'http://localhost/html/html5.html' click_on 'Play' find('#log').should
    have_text 'playing' end
  
```

fim

corredor de classe

incluir MeuMódulo

corrida definitiva

play_song end

fim

Corredor.novo.correr

No exemplo anterior, um dos testes do *Capítulo 4, Lidando com Ajax, JavaScript e Flash*, que verificava o comportamento de um componente de áudio HTML5, foi reescrito.

Em vez de usar cenários Cucumber, implementamos nosso próprio módulo e classe para executar o teste.

É importante requerer o arquivo Capivara DSL, pois contém todos os métodos Capivara que precisam ser "misturados". Neste exemplo, temos nosso próprio módulo Ruby e, crucialmente, dentro dele, o módulo Capybara relevante Capybara::DSL está incluído. Além disso, o módulo RSpec::Matchers também foi incluído, o que nos permite utilizar RSpec Matchers padrão (obviamente você não precisa usar RSpec; você pode escolher uma maneira diferente de afirmar o comportamento). Se você seguir esse padrão em seu próprio código, agora poderá misturar qualquer um dos métodos padrão do Capivara em seu próprio módulo ou métodos de classe.



Vale lembrar que, se você incluir módulos em uma classe base, todas as subclasses herdarão a capacidade de usar esses métodos de módulo. Isso seria útil, por exemplo, se você estivesse usando um padrão Page Object, onde a única classe que teria que incluir o módulo DSL seria a página base.

Usando a sessão diretamente

A outra opção para misturar o Capivara no seu código é usar a sessão diretamente, ou seja, você instancia uma nova instância do objeto da sessão e depois chama os métodos DSL nele.

O exemplo a seguir implementa o mesmo teste de antes, mas desta vez usando uma instância de sessão e levantando uma exceção simples se o conteúdo esperado não for encontrado:

```
requer 'capivara'
```

```
session = Capivara::Session.new :selenium
session.visit('http://localhost/html/html5.html')
session.click_on 'Play'
raise 'música não tocando' unless session.find('#log').text == 'jogando'
```

Se você estiver usando um modelo orientado a objetos para construir seus testes, precisará passar a instância da sessão ou encontrar uma estratégia apropriada para lidar com isso, pois não terá o benefício dos módulos misturados nos métodos DSL globalmente.

Tópicos Ninja

Capivara e estruturas de teste populares

O Capybara fornece integração pronta para uso com várias estruturas de teste populares; este não é um assunto que abordaremos em profundidade, simplesmente porque eles são muito bem abordados no LEIA-ME do Capivara, que pode ser encontrado em <https://github.com/jnifikas/capybara>.

Pepino Ao longo

deste livro, exemplos foram definidos no contexto do Pepino, portanto, você deve ficar satisfeito com a implementação da API do Capivara nas definições de etapas e fazer algumas configurações simples no arquivo env.rb, como definir o driver padrão . Existem algumas funcionalidades adicionais que o Capivara adiciona ao usar em conjunto com o Pepino, que vale a pena examinar.

A primeira é que a Capivara engancha no bloco Before do Cucumber da seguinte forma:

```
antes de fazer
  Capivara.reset_sessions!
  Capivara.use_default_driver end
```

Além de definir o driver padrão, isso faz uma chamada crucial para reset_sessions! e isso, por sua vez, invoca algum código no driver subjacente. No caso do Selenium, isso exclui todos os cookies para garantir que você inicie cada cenário sem poluição do anterior. Para Rack::Test, isso destruirá a instância do navegador, então uma nova será criada a cada vez; para quaisquer outros drivers, você precisará verificar a implementação da redefinição! para ver o que eles fazem.

Por fim, o Capybara também se conecta a qualquer cenário do Cucumber que você marcou com @javascript e muda automaticamente para o driver que você configurou para lidar com o JavaScript. Por exemplo, você pode usar Rack::Test como seu driver padrão, mas configurou o seguinte em seu arquivo env.rb :

```
Capivara.javascript_driver = :selenium
```

Nesse caso, qualquer cenário marcado com @javascript resultará no Capivara iniciando um navegador usando o Selenium como driver.

RSpec

Além do Cucumber, o RSpec é um dos frameworks de teste Ruby mais populares, prestando-se bem a testes de unidade, integração e aceitação, e usado dentro e fora do Rails.

Capivara adiciona os seguintes recursos ao RSpec:

- Permite misturar o DSL de acordo com suas especificações, adicionando require 'capybara/rspec' em seu arquivo spec_helper.rb
- Permite usar :js => true para invocar o driver JavaScript • Adiciona uma DSL para escrever testes de aceitação de "estilo de recurso" descritivos usando RSpec

Para obter mais detalhes sobre esses recursos, consulte o README, disponível em <https://github.com/jnicklas/capybara>.

Test::Unit Se você

estiver usando a biblioteca básica de teste de unidade do Ruby fora do Rails, então usar Capybara significa simplesmente misturar no módulo DSL via include Capybara::DSL, como você viu anteriormente.

Conforme observado no README, faz muito sentido redefinir a sessão do navegador em seu método de desmontagem , por exemplo:

```
desmontagem def
  Capivara.reset_sessions!
  Capivara.use_default_driver end
```

Se você estiver usando Rails, haverá outras considerações, como desativar os dispositivos transacionais, pois eles não funcionarão com o Selenium; novamente, o LEIA-ME do Capivara detalha esse comportamento completamente.

MiniTest::Spec MiniTest

é uma nova estrutura de teste de unidade introduzida no Ruby 1.9, que também tem a capacidade de suportar testes de estilo BDD.

O Capybara não possui suporte embutido para o MiniTest, porque o MiniTest não usa RSpec, mas usa seus próprios matchers. Há outra gem chamada capybara_minitest_spec (https://github.com/ordinaryzelig/capybara_minitest_spec), que adiciona suporte para esses matchers ao Capybara.

Tópicos Ninja

Interações avançadas e acesso direto ao motorista

Embora tenhamos abordado grande parte da API do Capivara, ainda existem algumas interações que não abordamos, por exemplo, passar o mouse sobre um elemento ou arrastá-lo.

Capivara fornece suporte para muitas dessas interações mais avançadas; por exemplo, uma adição recente (Capybara 2.1) aos métodos que você pode chamar em um elemento é hover:

```
find('#box1').
```

No Selenium, isso resulta em uma chamada para o método `mouse.move_to` e funciona para ambos os elementos usando a propriedade `hover` do CSS ou os métodos `mouseenter` / `mouseleave` do JavaScript . Outros drivers podem implementar isso de maneira diferente e, obviamente, em alguns, pode não ser suportado, seja onde o suporte a JavaScript for inexistente (Rack::Test) ou rudimentar (Celerity).

Você também pode emular arrastar e soltar usando a seguinte linha de código:

```
find('#mydiv').drag_to '#droplocation'
```

Novamente, o suporte ao driver provavelmente será irregular, mas é claro que isso funcionará bem no Selenium WebDriver.

Apesar de todos os sinos e assobios oferecidos pelo Capivara, ainda pode haver ocasiões em que você precise acessar a API que existe no driver subjacente, mas que não foi mapeada no Capivara. Neste caso, você tem duas opções:

- Chame o método nativo de Capivara em qualquer `Caybara::Elemento` e, em seguida, chame o método de driver
- Use `page.driver.browser.manage` para chamar os métodos de driver que não são chamado em elementos

Usando o método nativo

Se você deseja chamar um método no driver subjacente e esse método é chamado em um elemento DOM recuperado, você pode usar o método nativo .

Um bom exemplo é recuperar um valor de estilo CSS calculado. Os elementos no DOM podem obter propriedades CSS de duas maneiras; em primeiro lugar, há o **estilo inline**:

```
<p style="font-weight:bold;">Texto do parágrafo em negrito</p>
```

Essas informações podem ser facilmente recuperadas acessando o atributo style por meio dos métodos que discutimos no *Capítulo 2, Dominando a API*. A outra maneira pela qual um elemento obtém propriedades CSS é por meio de elementos de estilo ou folhas de estilo referenciadas por tags de link na página. Quando o navegador carrega as folhas de estilo e aplica estilos aos elementos DOM especificados, eles são conhecidos como **estilos calculados**.

O Capybara não tem API direta para recuperar o estilo computado de um elemento, o que provavelmente foi uma decisão de projeto deliberada, já que apenas alguns drivers suportariam isso. No entanto, o Selenium WebDriver tem esse recurso e é possível que você queira acessar essas informações.

Considere o código a seguir, no qual aplicamos uma propriedade CSS hover a um elemento div , para que, quando o usuário passar o mouse sobre o elemento, ele mude de cor.

```
<html>
  <cabeça>
    <title>Exemplos de foco</title> <style> .box
    { height: 200px; largura: 200px; margem:
      10px; cor de fundo: azul;

    } .box:hover {
      cor de fundo: verde;

    } </style> </
  head> <body>
  <div id="main">

    <div id="box1" class="box">
      </div>
    </div> </
  body> </
html>
```

Tópicos Ninja

As definições da etapa Pepino a seguir usam Capivara e Selenium WebDriver para afirmar que a cor mudou:

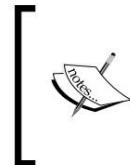
```
Quando (/^eu passar o mouse sobre um elemento cuja cor muda ao passar o mouse usando CSS$/)
visite 'http://capybara.local/html/chapter5/hover.html' find('#box1').hover
```

fim

Então(/^eu vejo a mudança de cor\$/) faça

```
find('#box1').native.style('background-color') .should == 'rgba(0, 128,
0, 1)'
fim
```

Aqui, o método de estilo no Selenium WebDriver é acessado por meio do método nativo do Capivara e o valor pode ser validado.



Isso destaca uma questão importante que vale a pena considerar ao verificar estilos computados. Diferentes navegadores podem relatar os estilos de forma diferente; por exemplo, alguns podem relatar uma cor como código hexadecimal e outros como código rgba; isso pode tornar seus testes frágeis ao serem executados em diferentes navegadores.



Acessando métodos de driver usando browser.manage

O outro caso de uso é quando desejamos acessar a funcionalidade fornecida no driver subjacente que não é mapeada pelo Capivara nem relacionada a um elemento específico na página.

Um bom exemplo disso é acessar informações de cookies. Houve um longo debate nos fóruns do Capybara e no rastreador de problemas do GitHub, sobre se o Capybara deveria expor uma API para getters e setters de cookies, mas o sentimento predominante sempre foi que isso não deveria ser exposto (sem dúvida, você não deve definir cookies em seus testes, pois isso está alterando o estado do aplicativo como um usuário nunca faria).

No entanto, é algo que você pode precisar fazer e, como o Selenium WebDriver e vários outros drivers oferecem suporte a essa funcionalidade, você precisará acessar os métodos do driver diretamente.

Considere esta página que define um cookie usando JavaScript:

```
<html>
<cabeça>
<title>Exemplos de cookies</title> <script>
document.cookie = 'mycookie=foobar'; </
script> </head> <body></body> </html>
```

As etapas a seguir simplesmente visitam a página e, em seguida, enviam para o console todos os cookies disponíveis na página atual:

```
Quando(/^visito uma página que define um Cookie$/) visite 'http://
localhost/html/cookie.html'
fim

Then(/^I can access the cookie using Selenium$/) do puts
page.driver.browser.manage.all_cookies end
```

Podemos acessar qualquer método no driver subjacente usando page.driver.browser.manage e, em seguida, o método que desejamos chamar; neste caso, chamamos o método all_cookies do Selenium WebDriver e a saída será a seguinte:

```
[{:name=>"mycookie", :value=>"foobar", :path=>"/html/
chapter5", :domain=>"localhost", :expires=>nil, :secure=>false}]
```

O Selenium WebDriver expõe uma API completa para acessar e configurar cookies. A documentação pode ser encontrada em http://rubydoc.info/gems/selenium_webdriver/0.0.28/Selenium/WebDriver/Driver.

Configuração avançada do driver

Até agora, definimos apenas o driver padrão ou o driver JavaScript usando um símbolo:

```
Capivara.default_driver = :selenium
```

É bastante provável que você precise ajustar a configuração do seu driver ou registrar várias configurações, que você pode selecionar em tempo de execução.

Tópicos Ninja

Um exemplo disso pode ser que você esteja executando testes em seu escritório e a rede corporativa esteja atrás de um Proxy HTTP (a ruína da vida de um testador). Se você estiver usando o Selenium WebDriver com o Firefox, poderá registrar uma configuração de driver personalizada no Capivara da seguinte maneira:

```
Capivara.register_driver :selenium_proxy do |app|
  profile = Selenium::WebDriver::Firefox::Profile.new
  profile["network.proxy.type"] = 1
  perfil["network.proxy.no_proxies_on"] = "capybara.local"
  profile["network.proxy.http"] = "cache-mycompany.com"
  profile["network.proxy.ssl"] = 'securecache-mycompany.com'
  profile["network.proxy.http_port"] = 9999
  profile["network.proxy.ssl_port"] = 9999
  profile.native_events = true
  Capivara::Selenium::Driver.new(app, :browser => :firefox, :profile => perfil)
```

fim

```
Capivara.default_driver = :selenium_proxy
```

Essa configuração usa a API do Selenium WebDriver para construir um perfil personalizado do Firefox, definir os detalhes do proxy programaticamente, registrar o driver com o nome :selenium-proxy e torná-lo o driver padrão.

Obviamente, as possibilidades aqui são infinitas e, em navegadores como Firefox e Chrome, a quantidade de opções que você pode personalizar chega a centenas, portanto, saber como defini-las é importante. Por exemplo, você pode usar esta técnica para criar perfis com JavaScript desativado ou Cookies desativados para garantir que seu aplicativo se comporte corretamente nesses casos.



Se você quiser saber mais sobre como usar diferentes navegadores e personalizar suas configurações usando o Selenium WebDriver, a documentação sobre as ligações do Ruby é o recurso mais útil (<https://code.google.com/p/selenium/wiki/RubyBindings>).

O ecossistema do motorista

O Capybara agrupa dois drivers, que como você sabe são Rack::Test e Selenium WebDriver. No entanto, o Capybara foi projetado de forma a facilitar a implementação de outros drivers pelos desenvolvedores e, de fato, existe um ecossistema saudável de drivers conectáveis, que oferecem alternativas interessantes para as duas opções integradas.

Capivara-WebKit

Praticamente todos os desenvolvedores que conheço que são apaixonados por automação de teste estão desesperados por uma coisa: um navegador headless com excelente suporte a JavaScript. Um navegador sem cabeça é aquele que é executado sem uma interface do usuário. Os navegadores headless geralmente executam testes mais rapidamente do que abrir um navegador real e facilitam a execução em ambientes de Integração Contínua (CI), onde muitas vezes um ambiente de janelas (MS Windows ou X11 no Linux) pode não estar disponível.

Você pode consultar os seguintes links para saber mais sobre o Capivara-WebKit:

- <https://github.com/thoughtbot/capybara-webkit>
- <http://qt.digia.com/>

Capybara-WebKit é um driver que envolve o QtWebKit e é mantido pelo pessoal da Thoughtbot. O projeto Qt fornece uma estrutura multiplataforma para a construção de aplicativos GUI nativos e, como parte disso, fornece uma implementação de navegador WebKit que se presta a implementações headless.

Há um pequeno problema com essa implantação específica do QtWebKit; você precisará instalar as bibliotecas do sistema Qt separadamente, e ainda há uma dependência do X11 presente nas distribuições do Linux, apesar do navegador ser headless.

poltergeist

Poltergeist é outro driver que, em segundo plano, usará o QtWebKit. A diferença aqui é que ele envolve o PhantomJS, o que traz algumas vantagens potenciais sobre o Capivara-WebKit.

Você pode consultar os seguintes links para saber mais sobre Poltergeist:

- <https://github.com/jonleighton/poltergeist>
- <http://phantomjs.org/>

Você ainda precisará instalar o PhantomJS como uma dependência do sistema, mas o projeto PhantomJS abordou alguns dos problemas em torno de tornar o QtWebKit puramente headless, então você não precisará do X11 e não precisará instalar todo o framework Qt, porque os pacotes do PhantomJS isso para você.

Capivara- Mecanizar

O Mechanize é um navegador headless implementado exclusivamente em Ruby e há muito tempo é um dos pilares da comunidade Ruby; ele usa Nokogiri em seu núcleo para fornecer acesso DOM e, em seguida, cria recursos de navegador em torno disso.

Você pode consultar os seguintes links para saber mais sobre o Capybara-Mechanize:

- <https://github.com/jeroenvandijk/capybara-mechanize>
- <https://github.com/sparklemotion/mechanize>

Capybara-Mechanize é um driver que permite que você use o Mechanize para executar seus testes. É uma opção muito poderosa, mas há alguns aspectos importantes a considerar, como:

- **Sem suporte a JavaScript:** Mechanize não contém um mecanismo de JavaScript; portanto, nenhum JavaScript em nenhuma de suas páginas será executado • **Nenhum mecanismo de renderização:** o Mechanize não contém um mecanismo de renderização; portanto, nenhum estilo computado será avaliado
- **Rápido:** como não está tentando avaliar JavaScript e fazer renderização gráfica, será super rápido

Os benefícios de usar o Mechanize podem não ser óbvios à primeira vista, e muito dependerá de como seu site é implementado. Para sites cuja funcionalidade é totalmente dependente de JavaScript, esta opção claramente não é viável; no entanto, se o seu site segue os princípios de "aprimoramento progressivo", onde o JavaScript é usado simplesmente para enriquecer a experiência do usuário, o Mechanize é uma ótima opção. Você pode implementar todos os testes funcionais principais usando o Mechanize, o que garantirá que eles sejam executados com latência mínima e serão muito menos frágeis do que o uso do Selenium e, em seguida, apenas implemente uma pitada de testes Selenium ou WebKit para testar os recursos dependentes do JavaScript.

Capybara-Celerity O driver final

que vale a pena considerar é o Capybara-Celerity, que envolve a biblioteca Ruby Celerity. Isso vale especialmente a pena se você estiver executando seu código de teste em JRuby (Ruby em execução na JVM), pois o próprio Celerity envolve a biblioteca Java HtmlUnit.

Você pode consultar os seguintes links para saber mais sobre a Capivara-Celeridade:

- <https://github.com/sobrinho/capybara-celerity>
- <https://github.com/jarib/celerity>
- <http://htmlunit.sourceforge.net/>

Voltando alguns anos, o HtmlUnit teria sido uma consideração séria para qualquer um que desejasse um navegador sem cabeça, na verdade era o navegador sem cabeça padrão para o projeto Selenium WebDriver. HtmlUnit é uma implementação Java pura de um navegador e usa Rhino (<https://developer.mozilla.org/en/docs/Rhino>) para executar JavaScript. Infelizmente, você pode descobrir que o suporte a JavaScript ainda falha ao tentar avaliar bibliotecas JS pesadas, pois o projeto não é mantido ativamente e acompanhar as demandas dos navegadores modernos não é realista para um projeto pequeno.

Se você está procurando uma alternativa sem cabeça para Mechanize, vale a pena dar uma olhada no Celerity, mas não confie nele para suporte a JavaScript.

Resumo

Este capítulo o levou da implementação de testes de Capivara com confiança em seu aplicativo para ser um ninja de Capivara.

Ao entender como usar o Capybara fora da zona de conforto do Cucumber, agora você pode usá-lo em praticamente qualquer ambiente e até mesmo escrever sua própria estrutura personalizada. Isso é importante mesmo se você usar o Cucumber, porque à medida que seus testes Cucumber crescem, você provavelmente desejará implementar alguns Page Objects e misturar Capivara::DSL nestes.

Outro aspecto importante de escrever testes eficazes é poder configurar o driver subjacente e acessá-lo diretamente quando necessário; não há nada de mágico nisso e significa que você pode aproveitar toda a potência do driver escolhido.

Por fim, apresentamos alguns drivers alternativos que devem abrir seu apetite e provar por que o Capivara é um framework tão poderoso. Você escreve seus testes uma vez e os executa em vários drivers compatíveis. Ganhar!

Machine Translated by Google

Índice

A

função addEventListener 67 **Ajax**
55, 56 **all_cookies** método 81
todos os métodos 30, 38, 39 **tag**
alt 23 **arquivo app.rb** 45 **assertion**
35 **JavaScript assíncrono** cerca de
55-58 localizadores 59 gotchas
60-62 matchers 59 métodos 59
valores de atributo verificando 40
objeto audioDebug 70 **elemento**
de áudio 65-67

B

browser.manage
usado, para acessar os métodos do driver 80,
81 **Bundler** sobre 7 gems, instalando com 7 URL
7 usado, para instalar o Capybara 10

C

Capivara
cerca de 43, 44, 56-58
CSS, usando 20, 21
instalando 5

instalando, Bundler usou 10
instalando, RubyGems usou 8
sessões, usando 75
URL 76
usado, módulos incluídos 74, 75
usando 73, 74
Capivara-Celeridade
cerca de 84
URL 84
Capivara-Mecanizar
cerca de 84
URL 84
capivara_minitest_spec
URL 77
Capivara fora do mundo de Cucumber
URL 73
Capivara-WebKit
cerca de 83
URL 83
caixas de seleção
25-27 verifique o
método 27 escolha
o método 27 atributo
de classe 61 click_on
método 28 teste de
componentes, in situ 68, 70
estilos computados 79
CSS
elementos, localizando com 19-21
usado, para Capivara 20, 21
Pepino
cerca de
76 instalando 11, 12
URL 11
Cucumber-Rails 13

D

Limpador de banco de dados
URL 53
elemento div 29, 40, 68, 79
Domain Specific Language (DSL) 5 driver
acessando 78 método nativo, usando 79, 80
usando 51, 52 **configuração do driver** 81,
82 **ecossistema do driver** Capybara-
Celerity 84 Capybara-Mechanize 84
Capybara-WebKit 83 Poltergeist 83 **métodos**
de driver acessando, browser.manage usado
80, 81

E

localização
de **elementos**, com localização
CSS 19-21, com visibilidade
XPath 19-21 33 **método**
avalia_script 70
API de interface externa
URL 63

F

field_labeled finder method 29 **fill_in**
method 17 **find_button finder method**
29 **find_by_id finder method** 29
finders about 59 refining 38, 39

find_field finder method 29
find_link finder method 29 **find**
method 29, 62
Firebug
URL 58
FirePathName
URL 22
Flash 63
FlashVars
URL 63

form.erb 45, 47
caixas de seleção
de formulários 25-27
botões de opção 25-27
envio 24

G

gems
instalando, com Bundler 7
instalando, com RubyGems 6
getElementById propriedade 65
getElementsByTagName propriedade 65
gotchas 60-62

H

tem_botão? 37
has_field? 37
has_link? 37
has_select? 37
has_table? 37
propriedade de foco 78, 79

“
atributo id 20, 23, 24, 27, 40 **estilo**
embutido 79 **elemento de entrada**
24

J

biblioteca jQuery
URL 58
eu
elemento de etiqueta 24
Carregar imagens botão 58
argumento do localizador 37

M

classe principal
49 matchers
cerca de 35-37, 59
refinado 38, 39
Miniteste::Especificação 77

módulos

 incluindo 74, 75
 método `mouse.move_to` 78
 múltiplas correspondências 30

N

 atributo de nome 17, 24, 27
 método nativo cerca de 78-80
 usado, para acessar o driver
 79, 80 navegação cerca de 22 botões,
 clicando em 22-24 links, clicando em 22-24

 Joya Nokogiri 7

P

 Objeto de página
 URL 74
 páginas
 testando 65-68
 Poltergeist
 por volta de 83
 URL 83

Q

 métodos de consulta 35, 38

R

 Prateleira
 URL 41
 Rack:: Teste
 cerca de 43, 44
 testes com 48-51
 Interface de rack 41-43
 botões de opção 25-27
 reset! método 76 result.erb
 47, 48 Rhino URL 85
 RSpec sobre 35-37, 77
 URL 17

Correspondentes

 RSpec usando 35

Ligações Ruby

 URL 82

Instalação do

 Ruby DevKit , URL 8

RubyGems

 cerca de 6

 gemas, instalando com 6
 usadas, para instalar Capivara 8

S

escopo 34

 marca de script 19
 Botão de pesquisa 17

Selênia

 instalando 11, 12

Selenium WebDriverName

 URL 81

sessão

 usando 75

instância de sessão 75

objeto de sessão 75

 aplicativo Sinatra arquivo
 app.rb 45 form.erb 45,
 47 result.erb 47, 48
 testando 45

situ

 componentes, testando 68-70
 elemento src 64 propriedade de

estado 70 definição de passo 11

estratégias combinando 30-32

 método de estilo 80 gems do
 sistema, instalando com Bundler
 7 gems, instalando com RubyGems
 6 preparando 6 bibliotecas do
 sistema instalando 8

T

método de desmontagem 77

Test::Unit 77

API testável

expondo 63-65

estruturas de teste

Pepino 76

Miniteste::Especificação 77

RSpec 77

Teste::Unidade 77

Afinar

URL 42

atributo de título 23

você

desmarque o método

27 desmarque o método 27

V

atributo de valor 23

visível? método 61

visita método 51

C

WAI-ARIA

URL 20

função wait_for 70

método wait_for 61

argumento within_fieldset 34

argumento within_frame(frame_id) 35 dentro do método 34 argumento within_table 34

argumento within_window 35

X

Elementos **XPath**, localizando com 19-21

Y

Pesquisa do YouTube 13-17



**obrigado por comprar
Teste de aplicativo com Capivara**

Sobre a Editora Packt

Packt, pronuncia-se 'packed', publicou seu primeiro livro "*Mastering phpMyAdmin for Effective MySQL Management*" em abril de 2004 e subsequentemente continuou a se especializar na publicação de livros altamente focados em tecnologias e soluções específicas.

Nossos livros e publicações compartilham as experiências de seus colegas profissionais de TI na adaptação e personalização dos sistemas, aplicativos e estruturas atuais. Nossos livros baseados em soluções fornecem o conhecimento e o poder para personalizar o software e as tecnologias que você está usando para realizar o trabalho. Os livros Packt são mais específicos e menos gerais do que os livros de TI que você viu no passado. Nosso modelo de negócios exclusivo nos permite trazer informações mais focadas, dando a você mais do que você precisa saber e menos do que você não precisa.

A Packt é uma editora moderna, mas única, que se concentra na produção de livros de qualidade e de ponta para comunidades de desenvolvedores, administradores e novatos. Para obter mais informações, visite nosso site: www.packtpub.com.

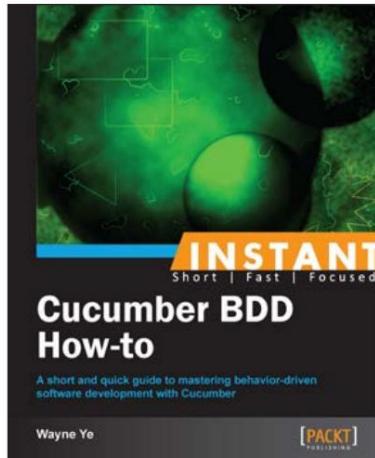
Sobre o Packt Open Source

Em 2010, a Packt lançou duas novas marcas, Packt Open Source e Packt Enterprise, a fim de continuar seu foco na especialização. Este livro faz parte da marca Packt Open Source, lar de livros publicados sobre software construído com base em licenças de código aberto e oferecendo informações a qualquer pessoa, desde desenvolvedores avançados até web designers iniciantes. A marca Open Source também executa o Open Source Royalty Scheme da Packt, pelo qual a Packt concede royalties a cada projeto de código aberto sobre cujo software um livro é vendido.

Escrevendo para Packt

Congratulamo-nos com todas as perguntas de pessoas interessadas em autoria. As propostas de livros devem ser enviadas para author@packtpub.com. Se a ideia do seu livro ainda está em um estágio inicial e você gostaria de discuti-la primeiro antes de escrever uma proposta formal de livro, entre em contato conosco; um de nossos editores de comissionamento entrará em contato com você.

Não estamos apenas procurando por autores publicados; se você tem fortes habilidades técnicas, mas não tem experiência em redação, nossos editores experientes podem ajudá-lo a desenvolver uma carreira de escritor ou simplesmente obter alguma recompensa adicional por sua experiência.



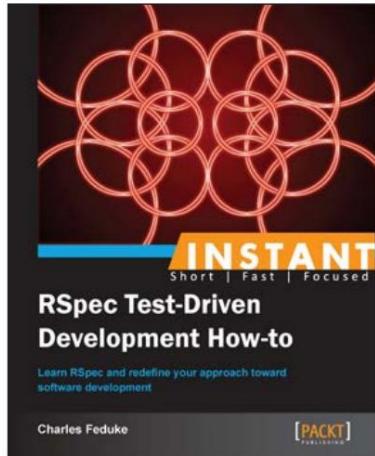
Como fazer o Instant Pepino BDD

ISBN: 978-1-78216-348-0

Brochura: 70 páginas

Um guia curto e rápido para dominar o desenvolvimento de software orientado a comportamento com o Cucumber

1. Aprenda algo novo em um instante! Um guia curto, rápido e focado que oferece resultados imediatos
2. Um processo passo a passo de desenvolvimento de um projeto real no estilo BDD usando Cucumber
3. Dicas profissionais para escrever recursos do Cucumber e passos
4. Apresenta algumas gemas populares e úteis de terceiros usadas com o Cucumber



Instant RSpec Test-Driven

Como fazer desenvolvimento

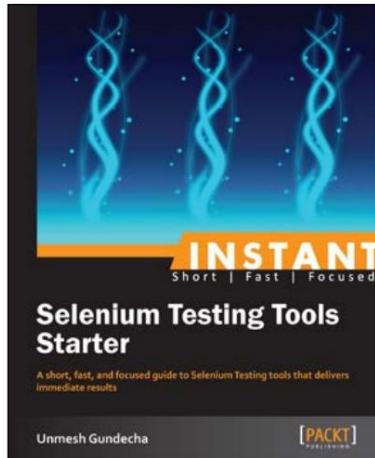
ISBN: 978-1-78216-522-4

Brochura: 68 páginas

Aprenda RSpec e redefina sua abordagem para o desenvolvimento de software

1. Aprenda algo novo em um instante! Um guia curto, rápido e focado que oferece resultados imediatos
2. Aprenda a usar RSpec com Rails
3. Exemplos fáceis de ler e grok
4. Escreva especificações idiomáticas

Verifique www.PacktPub.com para obter informações sobre nossos títulos



Ferramentas instantâneas de teste de selênio

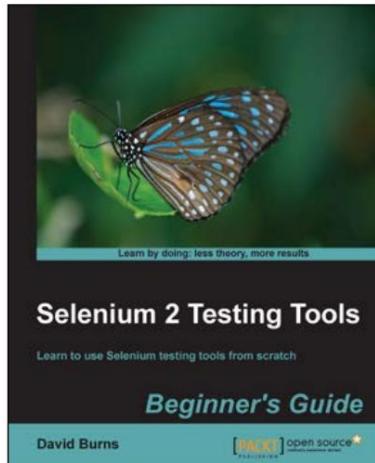
Iniciante

ISBN: 978-1-78216-514-9

Brochura: 52 páginas

Um guia curto, rápido e focado para ferramentas de teste de Selenium que oferece resultados imediatos

1. Aprenda algo novo em um instante! Um guia curto, rápido e focado que oferece resultados imediatos
2. Aprenda a criar testes web usando ferramentas Selenium
3. Aprenda a usar o padrão de objeto de página
4. Execute e analise os resultados do teste em uma plataforma fácil de usar



Ferramentas de teste de selênio 2:

Guia do iniciante

ISBN: 978-1-84951-830-7

Brochura: 232 páginas

Aprenda a usar as ferramentas de teste do Selenium do zero

1. Automatize os navegadores da web com Selenium WebDriver para testar aplicações web
2. Configure o Java Environment para usar o Selenium WebDriver
3. Aprenda bons padrões de design para testar aplicativos da web

Verifique www.PacktPub.com para obter informações sobre nossos títulos