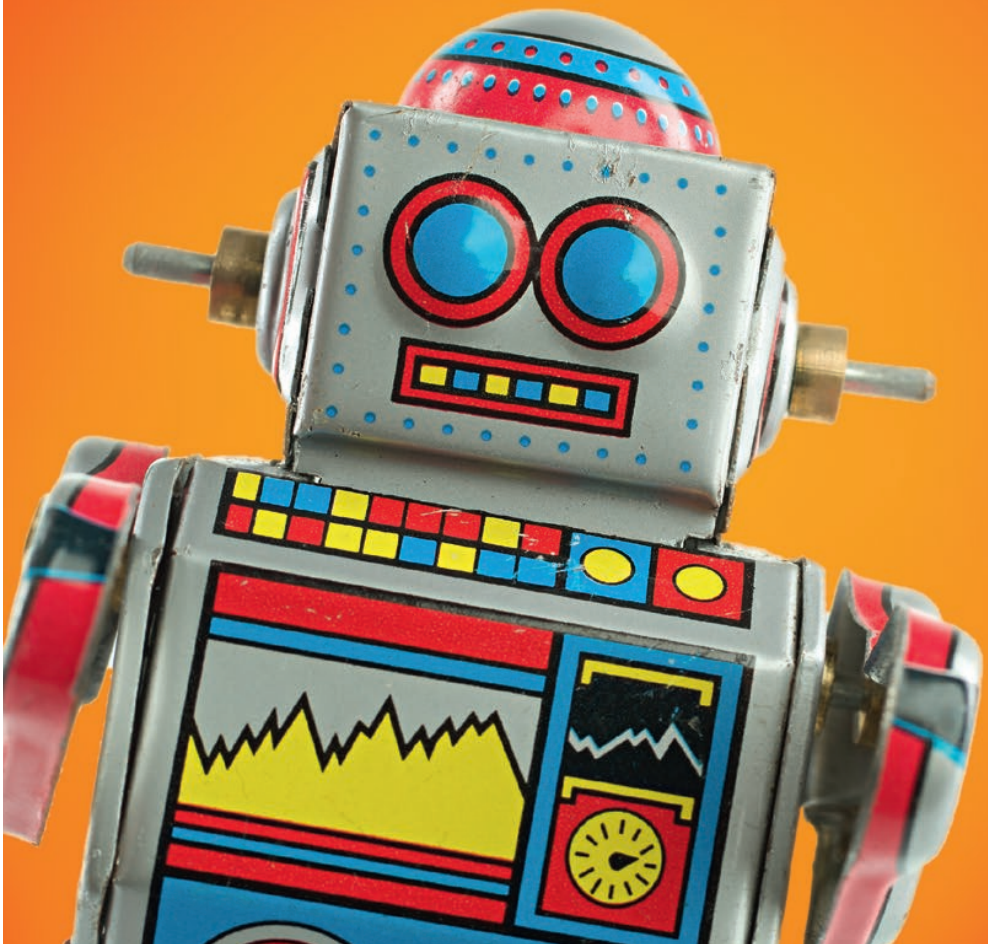
testing experience de

Das Magazin für Tester, Entwickler und Manager

Nr. 4 | Januar 2014
kostenlose Ausgabe

Testautomatisierung



Díaz Hilterscheid

Given When Test

Der Autor
Daniel Knott

Daniel Knott hat einen technischen Hintergrund in verschiedenen Programmiersprachen und Software Quality Assurance Tools. Nach seiner Ausbildung bei der IBM Deutschland GmbH studierte er an der Fachhochschule Wiesbaden Informatik mit dem Schwerpunkt Softwarequalitätssicherung. Anschließend arbeitete er bei der XING AG in Hamburg. In verschiedenen agilen Projekten (XING Suche, XING Empfehlungen) war er verantwortlich für das Testmanagement, für die Ausführung der Testfälle sowie für die Testautomatisierung. Als Team Lead Quality Assurance im XING Mobile- und API-Team entwickelte er voll automatisierte Testumgebungen für die Android- und iPhone-App. Aktuell arbeitet Daniel Knott als Software Test Manager bei der AOE GmbH in Wiesbaden, wo er für verschiedene Kunden das Software-Test-Management der Web- und Mobile-Anwendungen übernimmt.

Zusätzlich zu seiner Tätigkeit bei der AOE GmbH ist Daniel Sprecher auf verschiedenen agilen Konferenzen sowie Organisator der Software Test User Group Rhein Main (*www.stugrm.de*).

Sein XING-Profil: *https://www.xing.com/profile/Daniel_Knott*
Sein Blog: *http://www.adventuresinqa.com*
Twitter: *@dnlkntt*

Seit mehr als 10 Jahren gibt es die Verhaltensgetriebene Softwareentwicklung, besser bekannt als „Behaviour-Driven Development", kurz BDD [BDD01]. Seit der ersten Nennung der Technik von Dan North haben sich viele Sprachen und Tools diesen Ansatz zunutze gemacht, um Software früher und schneller zu testen. Schon während der Anforderungsanalyse der Software können die Aufgaben und die Ergebnisse der Anwendung textuell in „WENN – DANN"-Sätzen festgehalten werden, um sie später automatisiert gegen die fertig entwickelte Anwendung laufen zu lassen.

Dieser Artikel gibt einen groben Einblick, wie BDD mithilfe von Gherkin [BDD02], Cucumber [BDD03] und Capybara [BDD04] unter Windows genutzt werden kann, um eine beliebige Webanwendung gegen diverse Browser zu automatisieren.

Installation

Bevor mit der Entwicklung der Tests begonnen werden kann, müssen verschiedene Tools installiert sein.

- Ruby > 1.9 [INS01]
- Ruby Gems [INS02]
- QT4 Support für Windows [INS03]
- ChromeDriver [INS04]
- IEDriver [INS05]
- Ruby IDE oder Editor

Nachdem alles installiert ist, kann mit der Installation der Gemfiles begonnen werden. In einem Terminal müssen mit dem Befehl `gem.bat install` GEMNAME die benötigten Files installiert werden. Nachfolgende `gems` sollten installiert werden:

```
builder, bundler, bundler-unload,
capybara, capybara-screenshot, capybara-
webkit, childprocess, cucumber, diff-lcs,
ffi, gherkin, headless, json, mime-types,
mini_portile, multi_json, nokogiri,
rack, rack-test, rake, rspec, rspec-
core, rspec-expectations, rspec-mocks,
rubygems-bundler, rubyzip, rvm, selenium-
webdriver, websocket, xpath
```

Neues Projekt erstellen

Nach der Installation kann mit der Erstellung des neuen Projektes begonnen werden. Grundsätzlich bestehen Cucumber/Capybara-Projekte aus folgender Struktur:

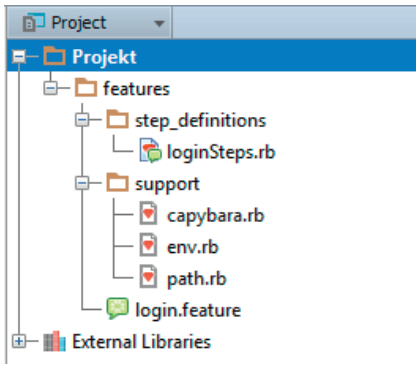


Abbildung 1. Struktur von Cucumber/Capybara-Projekten

Specifications Helper

Ist die dargestellte Ordnerstruktur angelegt, kann mit der Erstellung der Specifications Helper begonnen werden. Die Helper-Dateien werden im Ordner `support` abgelegt und dienen u. a. zur Konfiguration von Capybara, der Konfiguration der zu testenden Anwendung oder der Initialisierung der Testtreiber für verschiedene Browser. Des Weiteren können Funktionen wie das Erstellen von Screenshots im Fehlerfall dort implementiert werden.

Eine wichtige und zentrale Datei ist die `capybara.rb` (s. Abb. 2). In den ersten Zeilen dieser Datei werden die benötigten `gem files` eingebunden (Zeilen 1–5). In Zeile 7–9 wird mithilfe der Methode `getDriver` ein Kommandozeilenparameter `driver` definiert, mit dem die Tests bei der Ausführung gezielt gegen einen bestimmten Browser ausgeführt werden können. Der Befehl zum Starten der Tests sieht folgendermaßen aus: `cucumber.bat driver=chrome`. Wird der Parameter `driver` nicht gesetzt, wird der Standardtreiber `webkit` verwendet, um die Tests headless auszuführen.

In den Zeilen 12–31 werden die unterschiedlichen Browsertreiber initialisiert. Es können Browser auf dem lokalen Testsystem oder

```ruby
require 'capybara'
require 'capybara/dsl'
require 'capybara/cucumber'
require 'capybara/session'
require 'capybara-webkit'

def getDriver
  ENV['driver'] ? ENV['driver'] : :webkit
end

# Chrome Treiber
Capybara.register_driver :chromeLocal do |app|
  Capybara::Selenium::Driver.new(app, :browser => :chrome)
end

# Chrome Treiber im Selenium Grid
Capybara.register_driver :chrome do |app|
  Capybara::Selenium::Driver.new(app,
                                 :browser => :remote,
                                 :url =>  'http://SELENIUM_GRID:4444/wd/hub',
                                 :desired_capabilities => :chrome)
end

# Internet Explorer Treiber im Selenium Grid
Capybara.register_driver :internetExplorer do |app|
  Capybara::Selenium::Driver.new(app,
                                 :browser => :remote,
                                 # Remote Selenium 2 Grid URL
                                 :url => 'http://SELENIUM_GRID:4444/wd/hub',
                                 :desired_capabilities => :internet_explorer)
end

Capybara.default_selector =:xpath
Capybara.save_and_open_page_path = 'temp/capybara'
Capybara.run_server = true
Capybara.javascript_driver = getDriver.to_sym
Capybara.default_driver =  getDriver.to_sym
Capybara.default_wait_time = 2
```

Abbildung 2. Datei capybara.rb

aber auf einem Selenium Grid definiert werden (Zeile 12, Zeile 17). Am Ende der Datei wird festgelegt, wie Capybara die Anwendungselemente ansprechen soll; zur Auswahl stehen u. a. `css` oder `xpath`. In Zeile 36 und Zeile 37 wird der zuvor übergebene Browser gesetzt, der von Capybara verwendet werden soll.

In einer weiteren Datei wird ein `NavigationHelper` implementiert, mit dessen Hilfe die zu testende Anwendung spezifiziert wird. Es werden u. a. Pfade zu verschiedenen Bereichen der Anwendung festgelegt. Auch in diesem Helper kann mithilfe eines Kommandozeilenparameters die URL eingelesen werden, um sie im Testlauf zu nutzen. Dieser Parameter wird in Zeile 3–5 definiert. Wird kein `domain`-Parameter beim Starten der Tests gesetzt, wird automatisch `www.YOURAPPLICATION.com` genutzt, um die Tests gegen diese Domain auszuführen.

```ruby
module NavigationHelpers

  def current_domain
    ENV['domain'] ? ENV['domain'] : 'www.YOURAPPLICATION.com'
  end

  def path_to(page_name)
    domain = current_domain
    case page_name
      when /home/
        "http://#{domain}/"
      when /loginpage/
        "http://#{domain}/login"
      else
        raise "Can't find page with name \"#{page_name}\" to a path.\n" +
              "Please add a mapping in #{__FILE__}"
    end
  end
end

World(NavigationHelpers)
```

Abbildung 3. Implementierung eines NavigationHelpers

Die Methode `path_to` in Zeile 7–18 dient zum einfachen Aufrufen bestimmter Bereiche der Anwendung. Ist der übergebene Parameter in der Methode definiert, so wird die zuvor definierte Domain mit möglichen Parametern versehen und anschließend aufgerufen. Die `path_to`-Methode kann später in den Step Definitions genutzt werden, um einfacher und lesbarer mit der Anwendung zu interagieren. Solche Helper können je nach Bedarf mit beliebigen Funktionen erweitert werden, um die Anwendung einfacher bzw. strukturierter zu automatisieren.

Hooks

Ein weiteres Hilfsmittel, das während der Programmierung von Cucumber Tests genutzt werden kann, sind sogenannte Hooks [BDD05]. Cucumber stellt eine Reihe von Hooks zur Verfügung, die an jeder beliebigen Stelle während des Testlaufs aufgerufen werden können, um weiteren Code auszuführen. Es gibt u. a. Scenario Hooks, Step Hooks, Global Hooks oder Tagged Hooks. Hooks können dabei mit einer `setup()`- oder `tearDown()`-Methode verglichen werden. So kann z. B. mit einem Scenario Hook definiert werden, was mit der Anwendung, Datenbank oder anderen Systemen geschehen soll, bevor das Szenario ausgeführt wird. Jeder Hook kann dabei „vor (Before)“ oder „nach (After)“ einem Szenario ausgeführt werden. Hooks können an einer beliebigen Stelle im `support`-Ordner definiert werden. Das nachfolgende Codebeispiel zeigt einen Global Hook (Zeile 6–8). Dieser Hook öffnet nach dem Testlauf den generierten Testreport. In Zeile 9–10 wird ein Tagged Hook definiert. Alle Szenarios, die mit `@long` definiert sind, bekommen eine andere Standardwartezeit; in diesem Fall 30 anstatt 2 Sekunden.

```ruby
require 'rspec/expectations'
require 'test/unit/assertions'

World(Test::Unit::Assertions)

at_exit do
  system "open ./reports/report.html"
end
Before ('@long') do
  Capybara.default_wait_time = 30
end
```

Abbildung 4. Codebeispiel für einen Global Hook

So bieten auch Hooks ähnlich wie Specifications Helper eine tolle Möglichkeit, die automatisierten Tests dynamischer und robuster zu entwickeln, da auf verschiedene Situationen im Vorfeld reagiert werden kann.

Gherkin

Nachdem nun die Installation und das Setup von Capybara und Cucumber abgeschlossen sind, kann mit der eigentlichen Entwicklung der automatisierten Tests begonnen werden. Wer sich bereits über BDD informiert hat, kennt die Zweiteilung, in der die Tests geschrieben werden. Im ersten Schritt wird das Verhalten der Anwendung in einer domänenspezifischen Sprache, in diesem Fall Gherkin, geschrieben. Cucumber ist in der Lage, Gherkin zu interpretieren, um die Anweisungen in einem Test umzusetzen. Gherkin stellt ein Set an Befehlen zur Verfügung, die die Beschreibung der Anwendung sehr einfach macht. Die textuelle Beschreibung der Anwendung wird in sogenannten Feature Files vorgenommen [BDD06]. Die Dateiendung ist `NAME.feature`. Ein Feature File kann mit einer Klasse verglichen werden, die eine bestimmte Menge an Methoden, in diesem Fall Szenarios beinhaltet. Im ersten Schritt wird eine Beschreibung der Datei vorgenommen. Diese Beschreibung beginnt mit dem Bezeichner `Feature:`. Die Beschreibung soll einen schnellen Einstieg in die Funktionalität gewährleisten, die mit dieser Datei getestet werden soll.

Anschließend kann mit der Entwicklung des ersten Szenarios begonnen werden. Szenarios sollten immer nach dem Ansatz `Given When Then` geschrieben werden, um eine möglichst verständliche Beschreibung der Tests zu erhalten [BDD07]. Das nachfolgende Codebeispiel zeigt ein Szenario zum Ein- und Ausloggen an einem System.

```
Feature: As a user I want to be able to
log in.
  Scenario: Login and logout with a
  valid user
   Given I am on the "loginpage"
   When I enter a valid username and
      password
   And I click the login button
   Then I should be logged in
   When I click the logout button
   Then I must be logged out
```

Abbildung 5. Szenario zum Ein- und Ausloggen an einem System

Es ist sehr wichtig, dass die Beschreibung möglichst frei von technischen Details bleibt, um die Lesbarkeit und Verständlichkeit zu gewährleisten. Ebenfalls sollten Szenarios so geschrieben werden, dass sie unabhängig voneinander ausgeführt werden können. Das beschriebene Feature mit dem Szenario wird in der Datei `login.feature` gespeichert.

```
Given(/^I am on the "(.*?)"$/) do | pagename |
  visit path_to(pagename)
end

When(/^I enter a valid username and password $/) do
  fill_in("id_username", :with => "TestUserName")
  fill_in("id_password", :with => "SecretPassword")
end

Then(/^I click the login button$/) do
  click_button("Login")
end

Then(/^I should be logged in$/) do
  page.find("Welcome TestUserName").visible?
end

Then(/^I click the logout button$/) do
  click_button("Logout")
end

Then(/^I must be logged out$/) do
  page.find("Please login").visible?
end
```

Abbildung 6. Implementierung der Step Definition zum Ein- und Ausloggen

Nachdem nun ein Feature File mit einem ersten Szenario geschrieben wurde, kommt es zum zweiten Schritt, in dem die textuelle Beschreibung der Anwendung in Browserbefehle umgesetzt werden muss. Das Mapping von Text auf Befehle geschieht in sogenannten Step Definitions [BDD08]. Alle Step Definitions werden in einer Ruby-Datei mit der Endung `*.rb` im Ordner `step_definitions` abgelegt. Dabei muss keine Referenz zwischen Feature File und Step Definition File hergestellt werden, dies wird komplett von Cucumber/Capybara zur Laufzeit übernommen. Folgendes Codebeispiel zeigt die Implementierung der Step Definition zum Ein- und Ausloggen.

In diesen Steps wird das Mapping-Prinzip von Cucumber sehr gut sichtbar. Die zuvor geschriebenen Sätze werden mithilfe von regulären Ausdrücken nach Übereinstimmung geprüft. Existiert ein gültiger Ausdruck in den Step Definitions, wird der entsprechende Code ausgeführt. Des Weiteren wird sichtbar, dass die Schlagwörter Given When Then in den Step Definitions keinerlei Bedeutung haben und nur zur besseren Struktur und Lesbarkeit in den Feature Files dienen. Die Schlagwörter könnten in den Szenarios auch durch Sternchen (*) ersetzt werden.

Capybara stellt eine Reihe von Methoden zur Verfügung, die genutzt werden können, um mit der Anwendung zu interagieren, wie z. B. `click_button`, `click_link`, `fill_in`, `find`, `find_link`. Alternativ kann in den Step Definitions auch mit xpath- oder css-Selektoren gearbeitet werden, um Elemente auf der Seite eindeutig anzusprechen.

Besonders interessant ist die nachfolgende Step Definition:

```
Given(/^I am on the "(.*?)"$/) do
    | pagename |
  visit path_to(pagename)
end
```

Mit einem Given Step wird die Anwendung oder der Test in einen bestimmten Zustand versetzt, so kann z. B. eine Datenbankverbindung hergestellt oder wiederkehrende

Abbildung 7. Möglicher generierter Report

Schritte können zusammengefasst werden. In diesem Fall wird die Capybara-Methode `visit` verwendet, um eine bestimmte Seite der Anwendung zu öffnen. Weiter oben wurde in einem Specifications Helper die Methode `path_to(pagename)` definiert, um einfacher innerhalb der Anwendung zu navigieren. Diese Methode wird nun verwendet, die Zuordnung zum Helper übernimmt Cucumber/Capybara zur Laufzeit.

Sind alle Dateien gespeichert, kann mit dem Befehl `cucumber.bat feature/login.feature` der implementierte Test ausgeführt werden. Nach dem Testdurchlauf wird ein Report generiert, der aussehen kann wie in Abbildung 7 dargestellt.

Der Report enthält alle implementierten Features und Szenarios. Fehlgeschlagene Tests werden rot markiert und enthalten einen Hinweis, an welcher Stelle es zu einem Fehler gekommen ist. Im Fehlerfall kann ebenfalls durch einen Specifications Helper ein Screenshot an den Report angehängt werden.

Cucumber und Capybara bieten einen einfachen und schnellen Weg, zuverlässige Testsuiten für Webanwendungen zu entwickeln. Weitere Details und Anregungen zur Automatisierung mit Cucumber und Capybara sind in den Links & Referenzen enthalten.

Links und Referenzen

[BDD01] *http://dannorth.net/introducing-bdd/*

[BDD02] *https://github.com/cucumber/cucumber/wiki/Gherkin*

[BDD03] *http://cukes.info/*

[BDD04] *https://github.com/jnicklas/capybara*

[BDD05] *https://github.com/cucumber/cucumber/wiki/Hooks*

[BDD06] *https://github.com/cucumber/cucumber/wiki/Feature-Introduction*

[BDD07] *https://github.com/cucumber/cucumber/wiki/Given-When-Then*

[BDD08] *http://cukes.info/step-definitions.html*

[INS01] *http://rubyinstaller.org/*

[INS02] *http://rubyforge.org/frs/?group_id=126*

[INS03] *https://github.com/thoughtbot/capybara-webkit/wiki/Installing-Qt-and-compiling-capybara-webkit#windows*

[INS04] *http://chromedriver.storage.googleapis.com/index.html*

[INS05] *http://code.google.com/p/selenium/downloads/detail?name=IEDriverServer_x64_2.37.0.zip*