# Testing-Bibel

## Ziel der Bibel

Die Bibel ist das geltende Nachschlagewerk für das Test schreiben in unserem Projekt. Ihre Vorgaben sollten über das gesamte Projekt hinweg eingehalten werden. Sie beschreibt nicht nur was wann und wie getestet werden soll, sondern auch welche Tools, Hilfsmittel und Codekonventionen während dem Testing-Prozess genutzt werden.

## Die 10 Testing-Gebote

1. Du sollst keine Klasse ungetestet lassen!
2. Du sollst deine Tests schreiben bevor du die Funktionalität implementierst!
3. Du sollst den Code deinen Tests anpassen und nicht die Tests deinem Code!
4. Du sollst keine fehlschlagenden Tests ins VCS stellen!
5. Du sollst Regressionstests in Ehren halten und nur mit grösster Vorsicht anpassen!
6. Du sollst deine Tests verständlich und klar schreiben!
7. Du sollst dieses Dokument ehren und nicht die Testkonventionen brechen!
8. Du sollst möglichst realitätsnahe Daten zum Testen nutzen!
9. Du sollst Bugs vertestifizieren bevor du sie beseitigst!
10. Du sollst Tests reviewen wie du auch Funktionscode reviewst!

## Einführung

Das Testing soll in unserem Projekt eine zentrale Rolle spielen. Um die Funktionalität der Software in jedem Fall sicher zu stellen wird ein Test Driven Development angesteuert. Es wird also zuerst ein Test geschrieben, bevor wir die Stories implementieren.
Um das Testen so angenehm wie möglich zu gestalten nutzen wir einige Tools, darunter auch RSpec für unsere Ruby Anwendungen.

## Testresultate dokumentieren

Es müssen nicht alle Testresultate dokumentiert werden. Jedoch ist es besonders für manuell durchgeführte Tests wie Installations-, GUI- und Usability Tests wichtig, dass sie korrekt und detailliert dokumentiert werden. Diese Dokumentationen sollten mindestens folgende Punkte beinhalten:

- Was wird getestet?
- Wer testet es?
- Wann wird getestet?
- Welches Resultat wird erwartet? (Wenn möglich)
- Testablauf
- Was wurde festgestellt?
- Was schliessen wir daraus?
- Persönliche Eindrücke vom Tester
- Welches sind unsere daraus folgenden nächsten Schritte?
- Weiteres

Auch für automatisierte Tests sollten zum Teil Resultate dokumentiert werden, insbesondere bei Stress- und Datenbank-Tests, oder bei komplexeren Tests mit unerwarteten oder inkonsistenten Ergebnissen. Hier sollte mindestens folgendes festgehalten werden:

- Testklasse (evtl. kurze Beschreibung) & Autor
- Wann wurde getestet
- Welches Resultat erhalten wir?
- Was schliessen wir daraus?
- Wo gibt dies Probleme?
- Weiteres

Alle Testresultate werden im GIT im Ordner Dokumentation/Testing gespeichert, wobei für jeden Test ein neues Dokument erstellt wird.

## Tools

Um unsere Änderungen im Ruby-on-Rails Code zu testen verwenden wir, wie von SwissDRG vorgegeben, das Gem RSpec mit welchem mehrere Aspekte des Codes und seinen Ausgaben geprüft werden kann. Für ein kurzes Tutorial ist folgende Quelle gut geeignet: https://www.rubyguides.com/2018/07/rspec-tutorial/
Damit die Tests gut lesbar sind und auch fehlschlagende Regressionstests für uns, wie aber auch für den Kunden besser nachvollziehbar werden verwenden wir einige Codekonventionen. Diese versuchen wir möglichst an den bereits existierenden Tests anzulehnen.

**Codekonventionen**

Ein Test ist wie folgt aufgebaut:

```
describe User, type: :model do
  before(:all) do
    (2016..2017).each do |year|
      BillingYear.create(year: year)
    end
  end

  after(:all) do
    BillingYear.delete_all
  end

  let(:user) { User.create(username: 'user1234', password: 'password', password_confirmation: 'password', email: 'user@download.org') }

  context "no year granted for user" do
    it "should not have access to year" do
      expect(user.has_access_to_year(2015)).to be(false)
      expect(user.has_access_to_year(2016)).to be(false)
      expect(user.has_access_to_year(2017)).to be(false)
    end
  end
end
```

Auf Zeile eins ist der Beginn der Testklasse. Sie beschreibt entweder den Test einer Klasse (hier User) oder den zu testeten Prozess oder ein allgemeiner Zustand, welche als String beschrieben wird wie im folgenden Beispiel: `describe "password reset process", type: :feature do`

Die beiden Hooks before & after werden genutzt, falls für einige oder alle Testcases bestimmte Vorbedingungen erfüllt sein müssen, bzw. einige Teile wieder zurückgesetzt werden müssen. Für :all können bei Bedarf auch :each oder :suite eingesetzt werden. Des Unterschied ist jeweils zu welchem Zeitpunkt sie aufgerufen werden. Entweder nur einmal bevor die Testcases oder die Testsuite ausgeführt wird (:all bzw. :suite) oder vor/nach jeder Testmethode (:each), falls möglich sollte immer :all genutzt werden.

Durch das let werden Variablen definiert, welche schliesslich in der gesamten Testklasse genutzt werden kann.

Anschliessend werden die Testmethoden definiert. Sie bestehen aus einem Context und einem oder mehreren it, welche den Testfall kurz und pragmatisch beschreiben sollen, sodass sofort klar wird, was geprüft wird. Ein Context kann mehrere it in sich haben, falls dies sinnvoll ist.

Anschliessend wird der Test nach normalen RSpec Konventionen geschrieben. Die Methoden innerhalb eines it sollten kurz gehalten werden und stets nur ein Zustand/eine Änderung prüfen, sodass fehlschlagende Tests besser interpretiert werden können.

## Unit Tests

Unit Tests sind ein essentieller Teil unseres Development-Prozesses. Für jede geschriebene Klasse class.rb wird, am besten vorher, mindestens aber gleichzeitig eine Testklasse class_spec.rb geschrieben, welche im gleichen Package im spec-Ordner abgelegt wird. Diese Testklasse soll für jede Methode genügend Testcases beinhalten, damit nicht nur der normale Ablauf der zu testeten Methode kontrolliert wird, sondern auch Extremfälle oder Fälle die zu einem Fehler/einer Exception führen sollen abgedeckt werden. Erst wenn alle Tests erfolgreich durchgeführt werden können und sowohl die Klasse wie auch die Testklasse reviewt wurden, werden die Änderungen mitsamt der Testklasse ins VCS geladen.

Um eine Methode fundiert zu testen sollte man sich bevor sie geschrieben wird folgende Fragen stellen:

- Was soll die Methode erreichen?
- Welche Eingaben benutzt sie dafür?
- Wie geht sie mit unerwarteten Eingaben um?

Wenn diese Fragen beantwortet wurden können daraus zuerst Tests geschrieben werden, zuerst mindestens einer bei dem die erwarteten Eingaben gesendet werden. Bei diesen sollte kontrolliert werden, dass die Methode auch die erwartete Ausgabe generiert. Anschliessend sollten noch Extremfälle kontrolliert werden (NULL-Eingaben, falsche Objekte etc.) man muss sich überlegen, ob die Klasse eine Exception werfen soll oder eine spezielle Ausgabe dafür hat. Erst sobald diese Tests geschrieben wurden, startet die Implementation der Methode. Diese dauert solange an, bis alle Tests erfolgreich sind.

## Datenbank Tests

Dank den breitgefächerten Features von Ruby und dem PostgreSQL-Gem können Fehler oder Dissonanzen in der Datenbank relativ einfach erkannt und behoben werden, indem die Files unter db/migrate kontrolliert werden und Migrationsänderungen jeweils durch das PostgreSQL-Gem automatisch ins Version Control System gepushed werden.

## GUI Tests

Durch RSpec und bereits bereitgestellte Helper-Klassen können auch direkt GUI-Funktionen getestet werden.

```ruby
it "redirects to overview after login" do
  login_as user
  expect(page).to have_current_path('/')
end

it "renders overview" do
  login_as user
  visit '/'

  expect(page).to have_selector('div', text: 'Berechtigung')
  expect(page).to have_selector('div', text: 'Grouper Releases')
  expect(page).to have_selector('div', text: 'Spezifikationen')
end
```

Das login_as ist eine solche Helper-Methode mit welcher das Einloggen als bestimmte Rolle (user/admin) simuliert wird.
Es gibt auch die Methoden click_on *String*, welche den String versucht auf der Seite zu matchen und ihn anschliessend anklickt oder auch fill_in *String*, with: *Objekt* (z.B. fill_in 'Email', with: user.email), was versucht ein Feld, welches mit dem gegebenen String verbunden ist zu füllen mit dem definierten Objekt. Weitere solche Actions sind im actions.rb vom capybara-Gem beschrieben.
Mit visit kann ein bestimmter URL-Pfad besucht werden.
Die Matchers (have_selector, have_current_path, etc.), welche auch von capybara bereitgestellt werden (Cheatsheet: https://devhints.io/capybara ) können helfen schneller verschiedene Eigenschaften der Seite zu prüfen.

## Integrationstests

Das Integration-Testing wird automatisch durch Github durchgeführt. Es dürfen nur Änderungen gepushd werden, welche alle bisherigen und neuen Tests bestehen. Tests werden grundsätzlich nach ihrem ersten upload ins VCS nicht mehr verändert, einzig eine Erweiterung dieser darf noch stattfinden, was garantiert, dass die Integration neuer Funktionen reibungslos verläuft.

## Installationstests

## Stress-Tests

## Usability-Tests

## Umgang mit Bugs

Gefundene Bugs im System werden immer vor ihrer Behebung durch einen Testcase dokumentiert. Dabei wird eine Methode geschrieben (Entweder in einer sinnvollen bestehenden Testklasse oder in einer neuen Testklasse), welche den Bug reproduziert. Er sollte möglichst gleich ausgeführt werden, wie der reale Bug. Erst sobald diese Methode geschrieben wurde, ist der Bug zu beheben. Der Bug wurde entfernt, sobald der Test nicht mehr fehlschlägt. Ab diesem Zeitpunkt kann dank dem Test garantiert werden, dass sich der Fehler nicht erneut einschleichen kann, da er nun als Regressionstest standardmässig getestet wird.