Tag 1, Hands-On 1: Getting started with Ruby

Ziel des Hands-on

Ruby Syntax verstehen und in den Grundzügen schreiben können.

Aufgabe

1. Gib auf vier verschiedene Arten den String “Hello World!” viermal auf dem Bildschirm aus. Die unterschiedlichen Methoden sollten möglichst kreativ sein!
2. Eine Liste von Studenten, die Matrikelnummern den Namen der Studenten zuweist (mind. 4 Studenten in der Liste). Aus dieser Liste sollen die Namen sortiert ausgegeben werden.
3. Make an `OrangeTree` class. It should have a `height` method which returns its height, and a `one_year_passes` method, which, when called, ages the tree one year. Each year the tree grows taller (however much you think an orange tree should grow in a year), and after some number of years (again, your call) the tree should die. For the first few years, it should not produce fruit, but after a while it should, and I guess that older trees produce more each year than younger trees… whatever you think makes most sense. And, of course, you should be able to `count_the_oranges` (which returns the number of oranges on the tree), and `pick_an_orange` (which reduces the `@orange_count` by one and returns a string telling you how delicious the orange was, or else it just tells you that there are no more oranges to pick this year). Make sure that any oranges you don’t pick one year fall off before the next year. (“Learn to Program” von Chris Pine)

Ressourcen

- http://www.ruby-doc.org/
- Ruby Cheat Sheet

Shortcuts

Environment Setup

```
export PATH=/usr/local/bin:$PATH
export http_proxy=http://wwwproxy-gm.fh-koeln.de:8080
```

Strings

```
"Das ist ein String"
'Das hier auch'
```

Symbols

```
:das_ist_ein_symbol
```

Arrays

```
[1,2,3]
%w(eins zwei drei)
```

Hash

```
{ :key => 'value', :another_key => 'another_value' }
```

Ranges

```
0..4
```

Schleifen

```
4.times { |n| puts n }

while i < 10
  # do something
end

[1,2,3].each { |x| puts x+1 }

for i in 0..10 do |n|
  # do something with 'n'
end
```

Ausgabe

```
puts string
```

Allgemeines

- Ruby Source Dateien: `my_ruby_prog.rb`
- Ruby Interpreter starten: `ruby`
- Interactive Ruby Shell starten: `irb`
- Interactive Ruby Shell starten und Bibliothek laden: `irb -r bib.rb`
- Rückgabewerte: Eine Funktion gibt immer den zuletzt evaluierten Wert zurück, auch ohne Angabe des Schlüsselworts `return`.

Konventionen

- Klassen-/Modulnamen werden in CamelCase notiert
- Methodenamen werden mit Unterstrichen notiert (`find_by_name`)
- Methoden, die das Callerobjekt verändern haben einen Ausrufezeichen (`sort!`)
- Methoden, die true oder false zurückgeben haben ein Fragezeichen (`instance_of?`)

Tag 1, Hands-On 2: Getting started with Rails

Ziel des Hands-on

Ein Rails Projekt aufsetzen können und die Datenbank einrichten und migrieren

Aufgabe

1. Es soll die Projektstruktur für die “MyBlog” Applikation angelegt werden. Anschließend soll als erstes die Datenbank angelegt werden, anschließend soll das erste Modell der Applikation erzeugt werden. Als Hilfestellung dient das angefügte Flowchart der Blog-Applikation. Am Ende sollen die CRUD Funktionalitäten für das erzeugte Modell verfügbar sein und über den Browser ansprechbar sein. Mögliche Felder für einen Blogeintrag wären zum Beispiel:
    - Titel
    - Author (Name / Email)
    - Inhalt
2. Damit beim Aufruf der URL direkt die Liste aller Blogeinträge angezeigt wird, und nicht die “Welcome aboard” Nachricht von Rails, soll eine Default Route eingerichtet werden.
3. Mit Hilfe von `ActiveRecord::Migration` sollen die nötigen Datenbank-Tabellen angelegt werden.

Ressourcen

- http://www.ruby-doc.org/core
- http://api.rubyonrails.com
- http://wiki.rubyonrails.org/rails/pages/UnderstandingMigrations
- http://wiki.rubyonrails.org/rails/pages/UsingMigrations
- Ruby on Rails Cheat Sheet
- Rails Architektur

Shortcuts

Rails Projektstruktur initialisieren

```
rails <appname>
```

Generatoren und rake-Tasks

- Scaffolding: `script/generate scaffold <ModelName>`
- Server starten: `script/server`
- Migration erzeugen: `script/generate migration <name_der_migration>`
- Migration ausführen: `rake db:migrate`

Konventionen

- ActiveRecord Klassen sind immer im Singular
- ActionController Klassen sind immer im Plural

Wichtige Dateien

- Templates: `RAILS_ROOT/app/views/<ControllerName>/<action>.rhtml`
- Controller: `RAILS_ROOT/app/controllers/<ControllerName>_controller.rb`
- Modelle: `RAILS_ROOT/app/models/<ModelName>.rb`
- Konfiguration: `RAILS_ROOT/config/`
- Routing Konfiguration: `RAILS_ROOT/config/routes.rb`
- Statische Inhalte: `RAILS_ROOT/public`

Migrationen

Migrationsdateien: `db/XXX_<name>.rb` wobei XXX eine automatisch zugewiesene laufende Nummer ist.

Wichtige Methoden:

- `create_table / drop_table`
- `t.string :spalten_name` # Neue Spalte vom Typ `string` während des 'create_table' Blocks
- `t.integer :spalten_name` # Neue Spalte vom Typ `string` während des 'create_table' Blocks
- `add_column(table_name, column_name, type, options) / remove_column(table_name, column_name)`

Datentypen:

- Die Datentypen werden in die spezifischen DB Typen übersetzt.
- `integer, float, datetime, date, timestamp, time, text, string, binary, boolean`
- relevant sind für uns hauptsächlich `integer, string, text, datetime und boolean`

Wichtig In der `self.down` Methode müssen immer die Aktionen der `self.up` Methode rückgängig gemacht werden.

Tag 1, Hands-On 3: Validierungen

Aufgabe

1. Da gerade bei einem Blogsystem die Daten typischerweise aus einer Benutzereingabe in die Datenbank geschrieben werden, sollen diese Eingaben nicht ungesehen in der Datenbank landen. ActiveRecord bietet dafür sowohl fertige Validatoren an, als auch die Möglichkeit eigene Validierungen zu definieren. Aufgabe ist es also die Felder eines Blogeintrags entsprechend zu validieren, so dass keine unkorrekten oder gar bösartige Werte in der Datenbank landen.
2. Falls ein User eine nicht-valide Eingabe vornimmt soll er darüber informiert werden.

Ressourcen

- http://api.rubyonrails.com/classes/ActiveRecord/Validations.html
- http://api.rubyonrails.com/classes/ActiveRecord/Validations/ClassMethods.html

Wichtige Informationen

- Die Validierung findet statt bevor das Objekt gespeichert wird.
- Jedes ActiveRecord Objekt hat eine `validate` Methode, mit der die Validierung explizit getriggert werden kann.
- Scheitert eine Validierung, so wird die Fehlermeldung an das errors Hash des Objekts angehangen und kann in der View zur Fehlerausgabe verwendet werden.
- Typischerweise verwendet man das flash Objekt zur Ausgabe solcher Meldungen (das flash Objekt ist ein Hash, durch das Scaffolding wissen die Views bereits, dass es einen Key `notice` im `flash` Hash geben kann).

Einige Standardvalidierungen:

- validates_presence_of :name
- validates_uniqueness_of :key
- validates_format_of :email
- validates_size_of :password

Eigene Validierung:

```
def validate
  errors.add(birthday, "Who are you? Methusalem?") unless birthday < Time.parse("01-01-1850")
end
```

Tag 2, Hands-On 5: Relationen zwischen Modellen

Ziel des Hands-on

Relationen zwischen Modellen abbilden können

Aufgabe

1. Neben dem Modell `Article` ist das Modell `Comment` für die Applikation wichtig, da hierüber die Kommentare zu einem Artikel abgebildet werden. Die beiden Modelle gehen eine Relation ein, die in Rails mittels `ActiveRecord::Associations` abgebildet werden sollen. Dabei sollen gegebenenfalls die Datenbank überarbeitet werden, um Relationen abbilden zu können.
2. Relationen sollten validiert werden, um Relationsvorschriften gerecht zu werden. Ein Kommentar zum Beispiel nicht alleine existieren und muss immer zu einem Artikel gehören.

Ressourcen

- http://api.rubyonrails.com/classes/ActiveRecord/Associations/ClassMethods.html
- http://api.rubyonrails.com/classes/ActiveRecord/Validations.html

Shortcuts

Diese Aufgabe kann ganz ohne starten des Webservers gelöst werden und sollte auch so gelöst werden. Stattdessen soll rein über `script/console` gearbeitet werden. Hierdurch soll ein besseres Verständnis der internen Abläufe entstehen.

- `has_one`
  - 1:1 Beziehung auf Objektebene
  - Erweitert `Picture` um die Methode `thumbnail`
- `belongs_to`
  - Beschreibt die Tabelle, die den Foreign-Key in einer 1:n / 1:1 Beziehung hält
  - Erweitert `Address` um die Methode `person`
- `has_many`
  - 1:n Beziehung auf Objektebene
  - Erweitert `Person` um die Methode `addresses`
- `has_many_and_belongs_to`
  - n:m Beziehung auf Objektebene
  - Zwischentabelle, die lediglich in der Datenbank abgebildet wird (keine `ActiveRecord::Base` Klasse).
- `has_many :through`
  - n:m Beziehung
  - Zwischentabelle wird durch `ActiveRecord::Base` Klasse definiert
- Foreign-Keys müssen manuell angelegt werden
  - Konvention Klassenname (Singular) + `_id`
- Neben den Validierungen, die bereits im Handout 4 vorgestellt wurden, ist bei Assoziationen noch folgendes zu beachten:
  - Der Standard Validator `validates_associated :attribute` speichert das Objekt selbst nur dann, wenn auch alle seine assoziierten Objekte valide sind.
  - Entgegen der Rails Dokumentation sollte bei einer Assoziation sowohl geprüft werden, dass das assoziierte Objekt vorhanden, als auch, dass das assoziierte Objekt tatsächlich in der Datenbank vorhanden ist. Der Validator `validates_presence_of` muss demnach

sowohl auf das Objekt, als auch auf seine ID angewandt werden. Beispiel:

```
class Thumbnail < ActiveRecord::Base
  belongs_to :picture

  validates_presence_of :picture
  validates_presence_of :picture_id
end
```

Tag 2, Hands-On 6: Controller und Views

Ziel des Hands-on

1. Relationen auch auf der Controller und View Ebene umsetzen können.
2. Die ActiveRecord Methode `find` und `update` näher kennen lernen

Aufgabe

1. Die Relationen zwischen den Modellen haben Auswirkungen auf die Präsentations -und Controllerlogik. Deswegen müssen die Controller und View-Elemente angepasst werden. Aus der Einzelansicht eines Artikels sollte die Möglichkeit gegeben werden Kommentare zu einem Artikel anzuzeigen und neue Kommentare für diesen Artikel hinzuzufügen.
2. Die `ActiveRecord` Methoden `find` und `update` dienen als Abstraktionsmethoden für Datenbankoperationen. Es soll mittels der Methode `find` verschiedene Abfragen getätigt werden, die Auswirkungen auf die Artikel haben.

Ressourcen

- http://api.rubyonrails.com/classes/ActionController/Base.html
- http://api.rubyonrails.com/classes/ActionView/Base.html
- http://ar.rubyonrails.org/classes/ActiveRecord/Base.html

Shortcuts

Informationen zu der 1. Aufgabe

- Das Template zur Einzelansicht eines Artikels befindet sich in (app/views/articles/show.rhml)
- Ein Template zur Auflisten alle Kommentare sollte implementiert werden.
- Ein Template zum Anlegen eines Kommentar sollte implementiert werden.
- Der Zugriff der Kommentar-Templates sollte über die Einzelansicht eines Artikels erfolgen.
- Die zugehörigen Actions sollten sinnvoll benannt und an geeigneter Stelle implementiert werden.

Wichtige Methoden im Controller:

```
def show
  # Wertüberhabe über die params Methode
  @article = Article.find(params[:id])
end
```

- Templates innerhalb einer Action rendern oder an andere Actions redirecten
  - `render :template => 'name'`
  - `redirect_to :action => 'name'`

Wichtige Methoden in der View:

- `<% form_for :comment, @comment do |f| %>`
- `<%= f.text_field :author %>`
- `<%= submit_tag 'Save' %>`
- `<%= link_to 'Edit', edit_article_path %>`
- `<%= h @article.title %>`

Informationen zu der 2. Aufgabe

- Auf der Startseite des Blogs werden bestimmte Artikel angezeigt.
    - Es sollen die letzten 10 aktuellsten Artikel in der Liste ausgegeben werden.
    - Es sollen alle Artikel vom **heutigen Tag** ausgegeben werden. (Link auf der Startseite "Artikel von Heute")
    - Es sollen alle Artikel vom **gestrigen Tag** ausgegeben werden. (Link auf der Startseite "Artikel von Gestern")
    - Es sollen alle Artikel der **die älter** sind ausgegeben werden. (Link auf der Startseite "Artikel von letzter Woche")
- Find by id
    - `Person.find(1)  # returns the object for ID = 1`
    - `Person.find(1, 2, 6)  # returns an array for objects with IDs in (1, 2, 6)`
    - `Person.find([7, 17])  # returns an array for objects with IDs in (7, 17)`
    - `Person.find([1])  # returns an array for objects the object with ID = 1`
    - `Person.find(1, :conditions => "administrator = 1",  :order => "created_on DESC")`
- Find first
    - `Person.find(:first)  # returns the first object fetched by SELECT * FROM people`
    - `Person.find(:first,  :conditions => [ "user_name = ?", user_name])`
    - `Person.find(:first, :order => "created_on DESC",:offset => 5)`
- Find all
    - `Person.find(:all)  # returns an array of objects for all the rows fetched by SELECT * FROM people`
    - `Person.find(:all, :conditions => [ "category IN (?)", categories], :limit => 50)`
    - `Person.find(:all, :offset => 10, :limit => 10)`
    - `Person.find(:all, :include => [ :account, :friends ])`
    - `Person.find(:all, :group => "category")`

Tag 2, Hands-On 7: Partials

Ziel des Hands-on

Liste der Kommentare als Partials implementieren.

Aufgabe

1. Die Liste der Kommentare eines Artikels soll als Partial eingebunden werden. Dabei ist es wichtig das “Partial”-Konzept zu verstehen. Um Wiederholungen bei der Ausgabe der Kommentare zu vermeiden, sollte das Partial mittels einer Collection eingebunden werden.
2. Gemäß dem DRY-Prinzip sind Partials unerlässlich. Es sollen weitere Möglichkeitewn zur Erstellung von Partials gefunden und implemenitert werden.

Ressourcen

- http://api.rubyonrails.com/classes/ActionView/Partials.html

Shortcuts

Partial mit Übergabe einer lokalen Variable

```
<%= render :partial => "comment" %>
```
(Partial _comment.rhtml, lokale Variable person)

```
<%= render :partial => "account", :locals => { :account =>
@customer.account } %>
```
(Partial _account.rhtml, lokale Variable account)

Partial mit Übergabe einer Collection (Iteration z.B. über ein Array, Ausgabe für jedes Objekt im Array)

```
<%= render :partial => "comment", :collection => @articles.comments %>
```
(Partial _comment.rhtml, lokale Variable comment)

Partials mit anderen Controllern teilen

```
<%= render :partial => "user/login", :locals => { :user => @user.id } %>
```
(Partial /user/_login.rhtml, lokale Variable user)

Tag 2, Hands-On 8: Testing

Ziel des Hands-on

Die Grundlagen von automatisierten Tests verstanden haben und grundlegend anwenden können.

Aufgabe

1. Die bisher verwendeten Generatoren haben bereits einige Tests erzeugt. Jedoch werden diese, bedingt durch die durchgeführten Änderungen nicht mehr funktionieren. In diesem Hands-On sollen diese Test zum einen wieder ans Laufen gebracht werden, zum anderen sollen sie insofern erweitert werden, als dass die zusätzlichen Fähigkeiten ebenfalls getestet werden sollen. Dazu zählen dann also Tests, die überprüfen, dass ein Artikel immer einen Titel hat und ein Kommentar nicht ohne Artikel existieren kann.
2. Um die Tests durchführen zu können, müssen Testdaten vorliegen. Diese sollen in den jeweiligen Fixtures definiert werden.

Eine kleine Anmerkung

Die Methodik der testgetriebenen Entwicklung (TDD) entstammt der agilen Softwareentwicklung und dort Vorgehensmodellen wie Extreme Programming (XP). In der agilen Softwareentwicklung wird oft auf eine formale Spezifikation des Systems verzichtet, um schnell auf Änderungen reagieren zu können. Dies bedeutet jedoch nicht, dass überhaupt keine Spezifikation existiert! Die automatischen Tests bilden hier die Spezifikation. Daher ist es auch notwendig die Tests vor der eigentlichen Entwicklung zu schreiben. Sie definieren, wie eine Spezifikation, den Funktionsumfang der zu implementierenden Teile des Systems. Daher kommt die Aussage, dass Code der nicht getestet ist, auch nicht existiert.

Ressourcen

- http://api.rubyonrails.com/
- Ruby Test Cheat Sheet

Shortcuts

Testarten

- **Unit-Tests:** Testen Funktionalitäten auf Modellebene. Hier wird immer nur eine Modellklasse selbst getestet. Dazu zählen aber auch ob Relationen zwischen Modellen funktionieren.
- **Functional-Tests:** Testen ob die Controllerebene wie erwartet funktioniert. Ob Parameter richtig verarbeitet und die richtigen Objekte instanziiert werden, sind Fragen die hier beantwortet werden müssen. Ein Viewtesting ist hier nur sehr rudimentär möglich. In einem Functionaltest wird immer nur ein Controller getestet.
- **Integration-Tests:** Testen Abläufe innerhalb der Applikation. Hier werden verschiedene Controller im Zusammenspiel getestet. Es lässt sich simulieren wie sich ein Benutzer in der Applikation bewegt. Aber auch hier ist View-Testing nur rudimentär möglich.

Testcases

- Die Klassen der Testcases befinden sich im Ordner `test` und dort in den entsprechenden Unterordnern.

- Die Datei `test_helper.rb` wird von allen Tests geladen. Hier werden allgemeine Einstellungen vorgenommen (wie alle Fixtures zu laden und wie Fixtures geladen werden) und das Environment auf `test` gesetzt.
- Die Tests vererben von `Test::Unit::TestCase` (Unittests) oder `ActionController::TestCase`(Functionaltests, vererbt selbst aber auch von `Test::Unit::TestCase`). Dadurch sind zwei Methoden verfügbar, die überschrieben werden können: `setup` und `teardown`. Diese Methoden beinhalten Anweisungen, die vor bzw. nach jedem Test (also jeder Testmethode) auszuführen sind.
- Testmethoden beginnen immer mit `test_`, ansonsten werden sie nicht beim Testlauf ausgeführt.

Assertions

- siehe Cheat Sheet (die `Test::Rails` Assertions sind nicht verfügbar)
- Zusätzlich ist noch die Assertion `assert_difference(object, modifier, &block)` verfügbar, bzw. `assert_no_difference`

Fixtures

- In den Fixturesdateien zu jeder Modellklasse sind die Testdaten in YAML Notation spezifiziert. Sie entsprechen einer Art Objektserialisierung und daher können die selben Methodennamen verwendet werden, wie beim Erzeugen eines Objekts im Programm. Zu beachten ist, dass die Einrückungen in einer YAML Datei Leerzeichen sein müssen und keine Tabs (TextMate verwendet immer *SoftTabs*, setzt also Leerzeichen wenn man die Tabulatortaste drückt).
- Innerhalb der Tests kann auf die Testdaten über eine Methode die wie die Fixturedatei heisst zugegriffen werden, mit dem Namen des Datensatzes als Parameter (Bsp.: `articles(:article_for_test))`

Beispiel:

```
# Dieser Datensatz muss in der test/fixtures/articles.yml Datei stehen
# Auch gelten ganz normale Kommentare
article_for_test: # Name des Datensatzes
  title: Article for Pagination test
  author: dirk # Referenz auf einen Datensatz in der authors.yml Datei
  content: Lorem ipsum dolor sit amet, consectetur
  created_at:  # Fixtures durchlaufen auch den ERB Parser!
```

Ausführen der Tests

- `rake test:units` - Führt alle Unit Tests aus
- `rake test:functionals` - Führt alle Functional Tests aus
- `rake test:integration` - Führt alle Functional Tests aus
- `rake test` - Führt alle Tests aus

Beispiel

```
def test_should_have_valid_title
  article = Article.new(:title => '', :author => users(:jessie))
  assert !article.valid?
  assert article.errors.invalid?(:title)
  article.title = "Test Title"
  assert article.save
end
```

Tag 2, Hands-On 9: User Login

Ziel des Hands-on

- Die Stateless-Problematik von HTTP verstanden haben und mit der Session umgehen können.
- Einen User Login zumindest konzeptionell realisieren können.

Aufgabe

Natürlich soll nicht jeder beliebige Besucher auf unserem Blog einen Artikel veröffentlichen können. Diese Funktion soll nur bestimmten, registrierten Benutzern ermöglicht werden. Dazu muss:

1. Sich ein Benutzer einloggen können (zuerst muss er sich zwar registrieren, dieser Schritt soll hier jedoch nicht implementiert werden) und
2. Der Zugang zu bestimmten Teilen der Applikation muss gesperrt werden.

Weiterhin ist zu beachten, dass das Passwort nicht im Klartext gespeichert werden darf!

Ressourcen

- http://api.rubyonrails.com -> ActionController::Session und ActionController::Filter

Shortcuts

- Die Authentifizierung von Benutzern im System muss auf verschiedenen Ebenen implementiert werden:
    1. Das Modell `User` muss die Funktionen implementieren einen Benutzer anhand seiner Credentials zu identifizieren (Bsp: Email + Passwort)
    2. Der Controller muss den Zugang zu bestimmten Funktionen für nicht registrierte Benutzer untersagen
    3. Sobald sich ein Benutzer sich in das System eingeloggt hat, muss dieser Status irgendwo gespeichert werden.

Filter

In den Controllern können Filter definiert werden, die sicherstellen, dass eine bestimmte Aktion ausgeführt wird bevor oder nachdem eine Action ausgeführt wurde. Zusätzlich lassen sich die Actions einschränken, bei denen der Filter greift. Es gibt unterschiedliche Arten von Filtern, an dieser Stelle ist vor allem der `before_filter` von Interesse. Diese Klassenmethode erhält als Parameter eine Referenz in Form eines Symbols oder Strings auf eine Methode (die typischerweise als `private` deklariert wurde). Als weitere Parameter akzeptiert die Methode ein Hash mit den Actions die gefilter oder nicht gefiltert werden sollen.

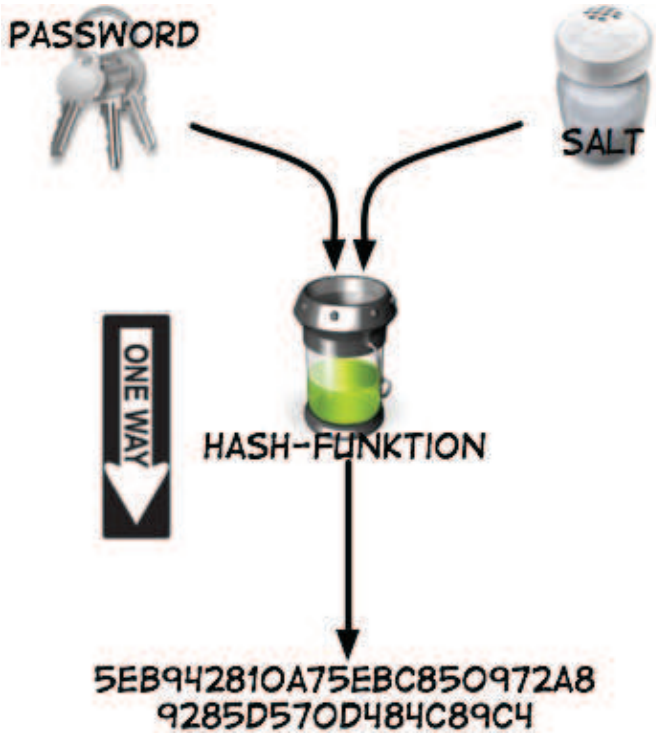
Bsp.:

```
before_filter :authenticated, :only => [:edit, :update, :new, :create, :destroy]
```

oder

```
before_filter :authenticated, :except => [:index, :show]
```

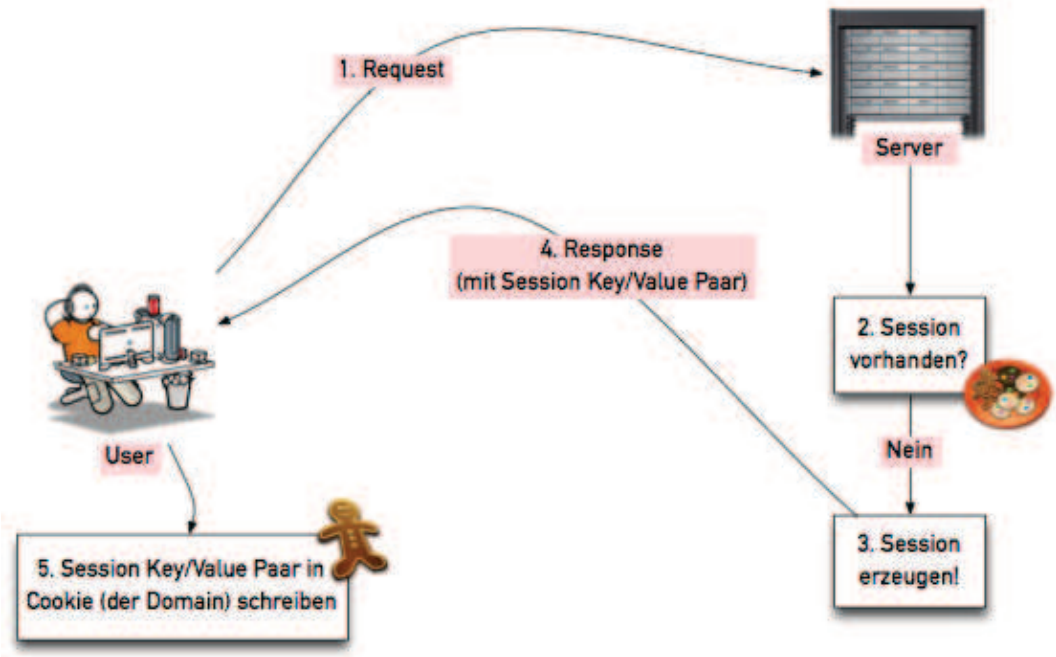
Hashfunktion



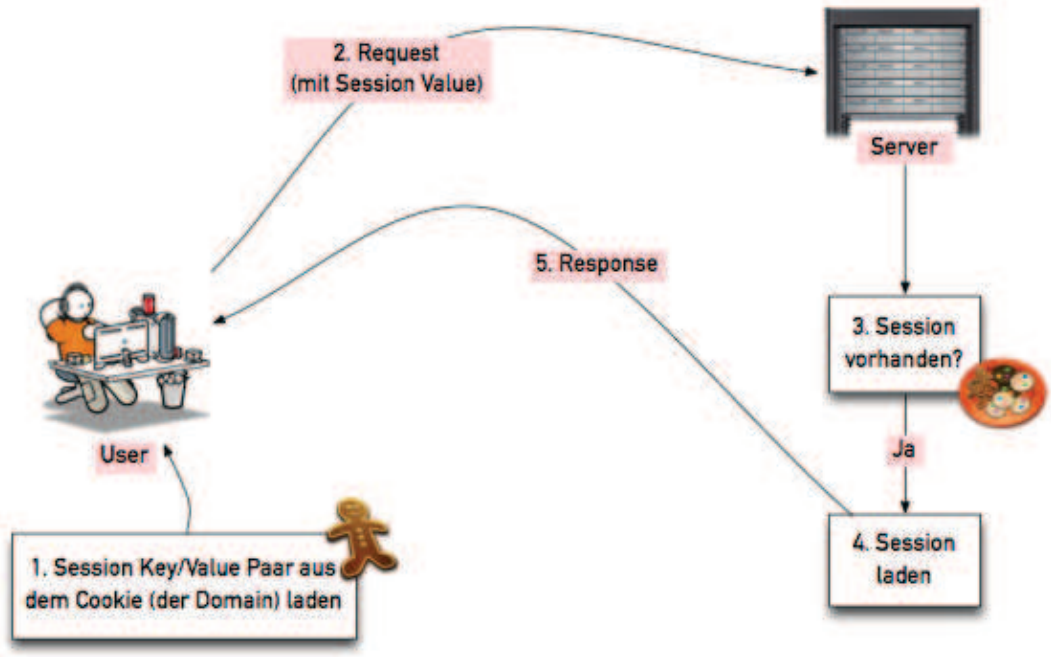
- Die Klasse `Digest::SHA1` und deren Methode `Digest::SHA1.hexdigest(str)` hashen eine String.

Sessions

Erster Request:



Zweiter Request:



- Die Methode `session` im Controller liefert Zugriff auf das Session Hash.
- In der Session sollten keine Objekte oder vertrauliche Daten gespeichert werden.
- Die Session sollte nicht im Cookie hinterlegt werden, jedoch für unseren Einsatz nicht so entscheidend.