Entwicklerdokumentation

Experimenteverwaltung - Erweiterung Buchungssystem (I2)

Richard Böhme

02. Juli 2021

Inhaltsverzeichnis

1. Einführung

Das Experimente Management System wird verwendet, um Physik Experimente zu verwalten und zu buchen. Es handelt sich bei der Anwendung um einen klassischen Rails Monolithen. Die nachfolgende Dokumentation soll helfen die Anwendung weiter zu entwickeln.

2. Struktur des Repositories

Das Repository besteht aus zwei wichtigen Ordnern.

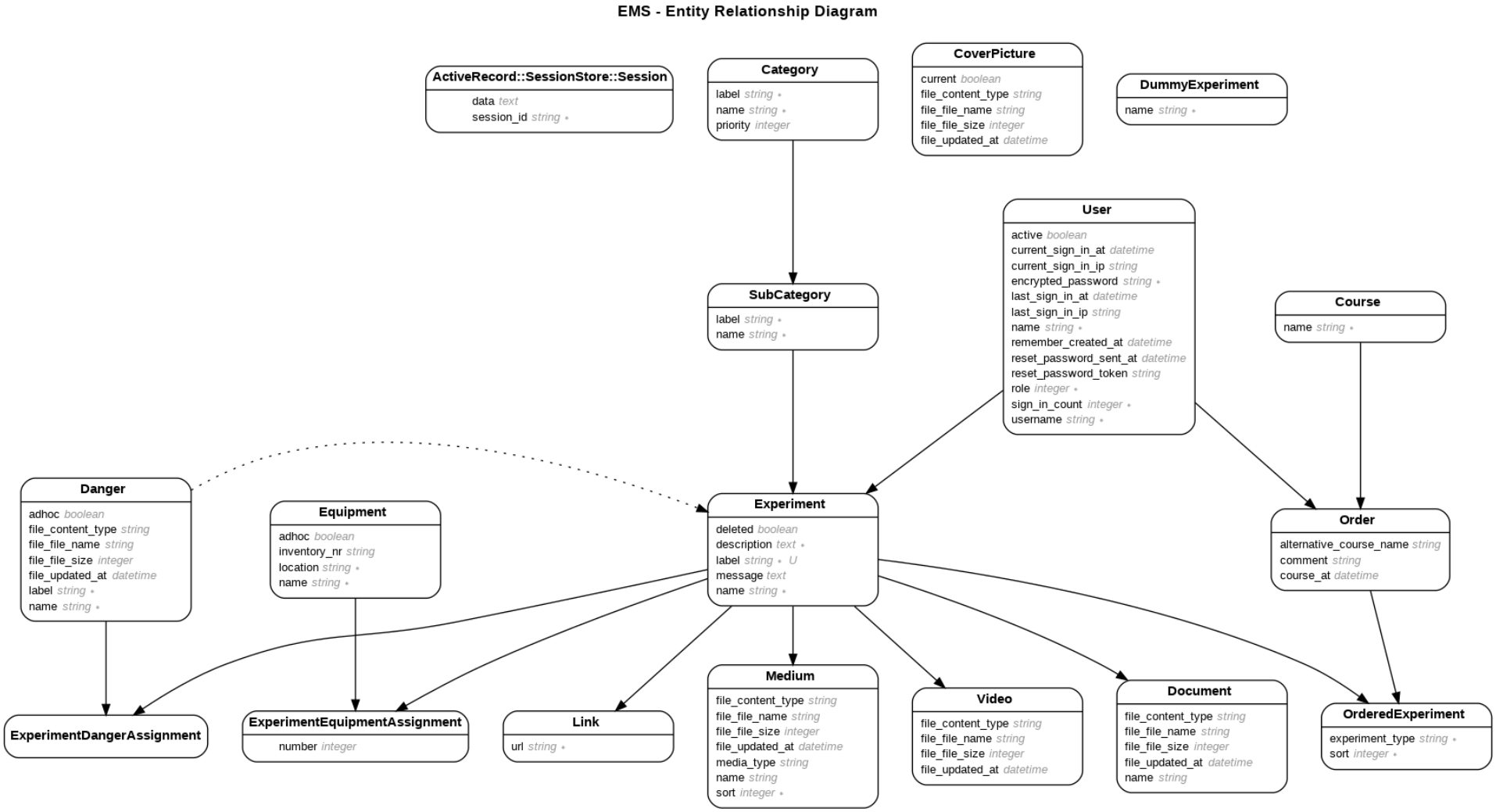
Der Ordner `Dokumentation` enthält jegliche Dokumentation zu dem Projekt. So z.B. das Architekturnotizbuch, die Anforderungsspezifikation oder Iterationspläne.

Der Ordner `src` beinhaltet den tatsächlichen Rails Quellcode.

3. Entity Relationship Diagram

Um einen schnellen Überblick über die bestehenden Relationen und deren Beziehungen zu geben, wurde ein Entity Relationship Diagram mit der Bibliothek rails-erd generiert.

Leider wird durch das Tool die polymorphe Beziehung zwischen der Klasse `OrderedExperiment` und den Klassen `Experiment` und `DummyExperiment` nicht korrekt abgebildet. Mehr Informationen zu polymorphen Beziehungen in Rails können hier gefunden werden.


EMS - Entity Relationship Diagram
ActiveRecord::SessionStore::Session
data text
session_id string *
Category
label string *
name string *
priority integer
CoverPicture
current boolean
file_content_type string
file_file_name string
file_file_size integer
file_updated_at datetime
DummyExperiment
name string *
User
active boolean
current_sign_in_at datetime
current_sign_in_ip string
encrypted_password string *
last_sign_in_at datetime
last_sign_in_ip string
name string *
remember_created_at datetime
reset_password_sent_at datetime
reset_password_token string
role integer *
sign_in_count integer *
username string *
SubCategory
label string *
name string *
Course
name string *
Danger
adhoc boolean
file_content_type string
file_file_name string
file_file_size integer
file_updated_at datetime
label string *
name string *
Equipment
adhoc boolean
inventory_nr string
location string *
name string *
Experiment
deleted boolean
description text *
label string * U
message text
name string *
Order
alternative_course_name string
comment string
course_at datetime
ExperimentDangerAssignment
ExperimentEquipmentAssignment
number integer
Link
url string *
Medium
file_content_type string
file_file_name string
file_file_size integer
file_updated_at datetime
media_type string
name string
sort integer *
Video
file_content_type string
file_file_name string
file_file_size integer
file_updated_at datetime
Document
file_content_type string
file_file_name string
file_file_size integer
file_updated_at datetime
name string
OrderedExperiment
experiment_type string *
sort integer *

4. Grundlegende Architektur - Architecture Notebook

4.1. Zweck

Dieses Dokument beschreibt die Philosophie, Nebenbedingungen, Begründungen, wesentliche Elemente und andere übergreifende Aspekte, die Einfluss auf Entwurf und Implementierung bezüglich der Realisierung eines in der Vision beschriebenen Buchungssystems für Physikexperimente haben. Die für die Architektur getroffenen Entscheidungen und Ihre Hintergründe werden hier festgehalten.

4.2. Architekturziele und Philosophie

Die Architektur muss eine Webanwendung zum Erstellen und Verwalten von Buchungen von Experiementen für Vorlesungen verwirklichen. Diese basiert auf einer bereits bestehenden Anwendung und ist somit als Weiterentwicklung zu betrachten. Aufgrund dessen muss die Architektur auf eben diese Weiterentwicklung bezogen sein und weiterhin durch Übersichtlichkeit, Eindeutigkeit und Verständlichkeit für eine hohe Nutzerfreundlichkeit sorgen.

Weitere Ziele, die die Architektur in Struktur und Verhalten erfüllen muss, sind:

- Gewährleistung einer hohen Nutzungsdauer - das System soll über einen langen Zeithorizont genutzt werden und muss für eine variierende Nutzergruppe ausgelegt sein
- es muss Kompatibel für zu erwartende Änderungen sein, bspw.:
  - die Einführung einer Kopierfunktion für vergangene Buchungen
  - die freie Auswahl von betrachteten Zeiträumen bei der Statistik
  - die Änderung der Entwicklerbasis - das System soll zu einem unbestimmten Zeitpunkt von einem anderen, dem jetztigen nicht bekannten Entwicklerteam weiterentwickelt werden

4.3. Beschränkungen, Annahmen und Abhängigkeiten

- **Grundannahme:**
  - das bestehende System wird weiter-, nicht neuentwickelt
  - ein System zur Verwaltung von Physikexperimenten und eine zugehörige Datenbank bestehen bereits
- Ruby ist als Programmiersprache zu verwenden
- Vorgabe: Rails-Framework
- es wird angenommen, dass die VM ausreichend Ressourcen bietet, um die Webanwendung in ihrem zukünftigen Ausmaße zu betreiben
- der Server nutzt das Betriebssystem Linux
- SQLite ist als Datenbankmanagementsystem zu verwenden

- die Nutzung von NGINX und Passenger als Webserver werden vom bestehenden System verlangt

4.4. Architektur-relevante Anforderungen

Über die systemweiten funktionalen und nichtfunktionalen Anforderungen kann im Dokument zu den systemweiten Anforderungen nachgelesen werden.

Anforderung	Systemkomponente	Architekturmechanismen
UC01	• Webanwendung • Weboberfläche • Datenbank • Experiment	• Sicherheit • Persistenz • Kommunikation • Eingabelogik • Archivierung • Session-Management • Error Management
UC02	• Webanwendung • Weboberfläche • Buchung • Datenbank • Experiment	• Zugriffsschutz • Kommunikation • Eingabelogik • Informationsaustausch • Error Management
UC03	• Webanwendung • Weboberfläche • Buchung • Datenbank • Experiment	• Zugriffsschutz • Kommunikation • Eingabelogik • Error Management
SWFA-1	• Webanwendung	• Zugriffsschutz
SWFA-2	• Datenbank	• Persistenz
SWFA-3	• Webanwendung	• Verschlüsselung
NFAU-1, NFAU-2	• Weboberfläche	• Eingabelogik
NFAR-1	• Webanwendung	• Session-Management

Anforderung	Systemkomponente	Architekturmechanismen
NFAS-1	• Webanwendung	• Informationsaustausch
NFAS-2	• Weboberfläche	• Kompatibilität

4.5. Entscheidungen, Nebenbedingungen und Begründungen

1. Weiterverwendung folgender Aspekte aufgrund zeitlicher Aufwandsbegrenzung:
    a. Datenbank
    b. Framework
    c. Sprache
    d. Frontend
    e. visuelles Design
    f. Bibliotheken
    g. Backup-Routine
    h. Klassen
    i. Objekte
    j. Konfigurationen
2. Weiterverwendung des bis jetzt genutzten Test-Frameworks (RSpec) und der bis jetzt existierenden (teilweise fehlerhaften) Testcases und damit einherkommende Fehlerbehebung
    a. Grund: beim Review des alten Testquellcodes wurde festgestellt, dass ca. 30% der Tests mit einem Fehler abschließen, weshalb der Gedanke zum Testframework Minitest zu wechseln diskutiert wurde, da dies schon bekannt ist - Mehraufwand durch Umbauen der alten Tests ist jedoch vermutlich größer als Einarbeitungszeit in unbekanntes Framework Rspec; Risiko beim Frameworkwechsel, dass Aspekte übersehen werden
3. Verwendung von kostenlosen Bibliotheken (Open Source) und Lizenzen
    a. Grund: dringende Vermeidung von Kosten
4. Verwendung von Ruby/Rails
    a. Wiederverwendbarkeit = schnelle Entwicklung (adressiert Risk No. 4)
    b. Weniger komplex; für alle Entwickler in gegebener Zeit erlernbar sofern Kompetenz noch nicht ausreichend (adressiert Risk No. 5)
5. Festlegung einer funktionalen Grenze: Kopierfunktion als Zukunftsvision
    a. Grund: Architekturbezogene Komplexität mit Zeitbudget nicht realisierbar
6. Einführung von Quellcodekommentierung
    a. Grund: Arbeitserleichterung des Nachfolge-Entwicklerteams
7. Verwendung von Drops-Downs, Datepicker etc. um entsprechend der Anwenderanforderungen

für intuitiv bedienbare Elemente zu sorgen

a. Vermeidung von PopUps um der Anwenderanforderung gerecht zur werden

4.6. Architekturmechanismen

1. **Archivierung**
   - Zustand: Implementation (bestehendes System)
   - Zweck: Daten dürfen im Normalbetrieb bei möglichen Systemausfällen oder Anwendungsfehlern nicht verloren gehen
   - Eigenschaften/Attribute: Backups werden im Moment täglich von Datenbank und Bilddateien angelegt, wobei stets nur die letzten 5 Backups behalten werden
   - Funktion: Automatisches Backup von Datenbank und Bilddateien wird auf dem Fakultätsserver angelegt
2. **Kommunikation**
   - Zustand: Implementation (Framework wurde festgelegt)
   - Zweck: Kommunikation zwischen Datenbank und Webanwendung
   - Eigenschaften/Attribute: lückenlos, synchron
   - Funktion: z.B. zwischen DB/Buchungssystem
3. **Error Management**
   - Zustand: Analysis
   - Zweck: Information des Anwenders bei einem unerwarteten Fehler
   - Eigenschaften/Attribute: Standardmäßig im Browser
   - Funktion: Rails und der Webservice geben bei einem unerwarteten Fehler eine Standard-Fehlerseite aus
4. **Informationsaustausch**
   - Zustand: Analysis
   - Zweck: Übergabe von Informationen zwischen zwei Akteuren (Menschen oder Systemen, auch untereinander)
   - Eigenschaften/Attribute: fehlerfrei, verständlich, vollständig
   - Funktion:
     - “Übergabe” der gebuchten Dozentenwoche: Dozent→ Admin durch System
     - Kann auch Kommentare im Quellcode beschreiben (=zwischen Entwicklerteams)
5. **Kompatibilität**
   - Zustand: Analysis
   - Zweck: System muss auf allen (in den SWRs (siehe SWR)) erwähnten Browsern funktionieren
   - Eigenschaften/Attribute: gegeben oder nicht gegeben

- Funktion: Sicherstellen der Benutzbarkeit

6. **Eingabelogik**
   - Zustand: Design
   - Zweck:
     - Designvorschriften für schnelles Nutzerverständnis
     - Folge von Eingaben muss in eingegrenztem Schema ablaufen
   - Eigenschaften/Attribute: einfach, nicht fehleranfällig, klar strukturiert
   - Funktion:
     - über Drop-Down-Menüs, Date-Picker, Freitextfelder, Ausgrauungen werden die Auswahlmöglichkeiten begrenzt und kontrolliert
     - keine Formulare in Popups
7. **Persistenz**
   - Zustand: Implementation (bestehendes System)
   - Zweck:
     - Daten müssen für Admin verfügbar sein
     - Daten müssen für Journal und Statistik existieren
     - Daten dürfen nicht verschwinden (Chronologie der Statistik)
   - Eigenschaften/Attribute: Datenbank ist eine SQLite-DB
   - Funktion:
     - Bestellungen werden in der Datenbank gespeichert
     - Datenbank muss ausreichend Felder für alle Daten bereitstellen
8. **Session-Management**
   - Zustand: Analysis
   - Zweck: Buchungsdaten bleiben während einer validen Anmeldung bestehen
   - Eigenschaften/Attribute: semipersistent
   - Funktion: Session-Token wird bei Anmeldevorgang vergeben und dient als Zuordnungsschlüssel zur Sitzung und ihrer Daten
9. **Datenmigration**
   - Zustand: Analysis
   - Zweck: Daten dürfen bei der “Rückübertragung” von der VM nicht verloren gehen → Schutz vor Datenverlust (adressiert Risk No. 6)
   - Eigenschaften/Attribute: vollständig, fehlerfrei
   - Funktion: Ausführung von Migrationsskript(en)
10. **Zugriffsschutz**
    - Zustand: Analysis

    - Zweck:
        - Authentifizierung
        - Login
        - aktive Sessions
        - → Datenschutz (adressiert Risk No. 1)
    - Eigenschaften/Attribute: jeder Nutzer hat nur spezifischen Zugriff
    - Funktion: bei Anmeldung Passworterhalt für Dozenten, bei Login Eintreten der Zugriffsbeschränkungen
11. **Verschlüsselung**
    - Zustand: Implementation (bestehendes System)
    - Zweck: Nutzerdaten werden sicher gespeichert
    - Eigenschaften/Attribute: sicher, nicht auslesbar
    - Funktion:

4.7. Wesentliche fachliche Abstraktionen

- **Experiment:** Teil der Buchung - Das Objekt wird beim Buchungsvorgang aus der Datenbank bezogen. Das Objekt wird in Dozentenwoche angezeigt.
- **Nutzer (Vererbung):**
    - **Admin:** Anwender-Rolle - Erstellt und bearbeitet Experimente. Verwaltet Rechte der Dozenten. Betrachtet alle Dozentenwochen. Betrachtet Statistik.
    - **Dozent:** Anwender-Rolle - Führt Buchung durch. Betrachtet seine Dozentenwoche. Betrachtet Journal.
- **Buchung:** Wird vom Dozenten erstellt. Beinhaltet ein oder mehrere Experimente. Besitzt zusätzliche Daten.
- **Dozentenwoche:** Ausgabe - Wird von Admin und/oder Dozent betrachtet. Enthält alle Buchungen eines Dozenten innerhalb des bestimmten Zeitabschnitts=einer Woche.
- **Journal:** Ausgabe - Wird vom Dozenten angefordert. Auflistung aller getätigten Buchungen eines Dozenten. Wird vom Dozenten betrachtet.
- **Statistik:** Ausgabe - Wird vom Admin angefordert. Wird vom Admin betrachtet. Enthält Anzahl von allen in einer bestimmten Periode gebuchten Experimenten, sortiert nach spezifischen Kategorien.

4.8. Schichten oder Architektur-Framework

4.8.1. Model-View-Controller (MVC)

- ergibt sich durch Rails
- **Model** = Schicht der Geschäftslogik und ihrer Anwendung (siehe Domänenmodell)

- **View** = Schicht der HTML-Verarbeitung zur Darstellung der Ressourcen
- **Controller** = Schicht der Requestverarbeitung durch http-Anfragen

4.9. Architektursichten (Views)

4.9.1. Logische Sicht (C4-Modell)

Context

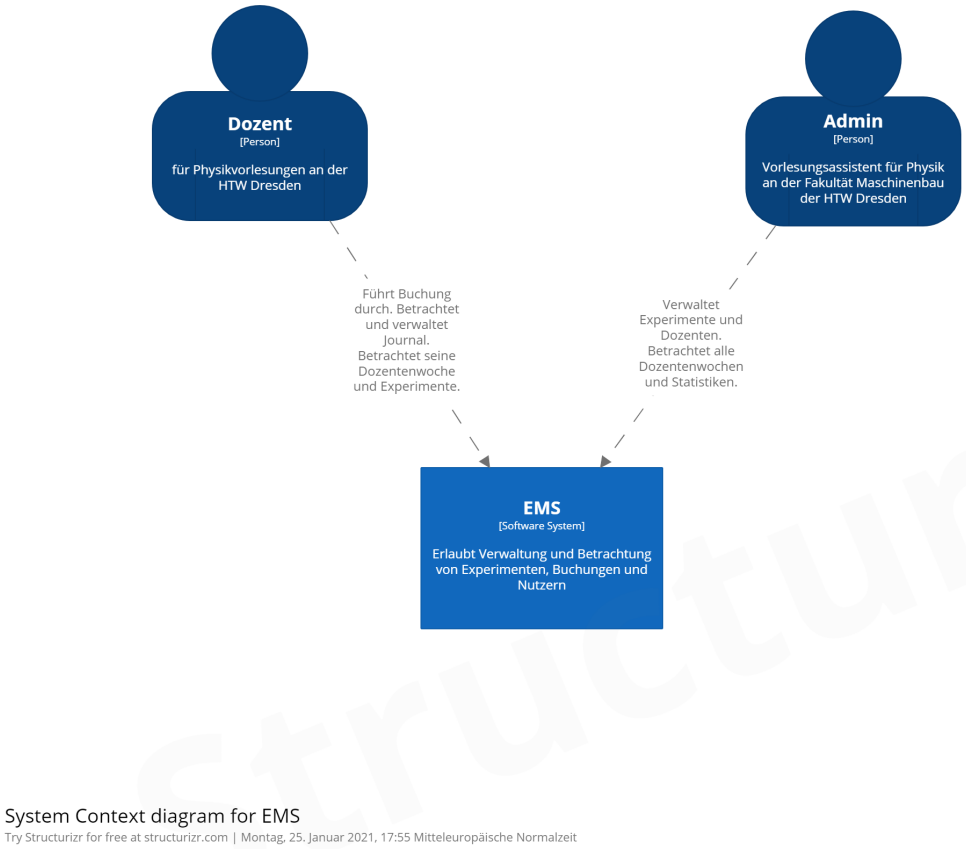


Abbildung 1. Context-Diagramm

Abbildung 2. Context-Diagramm - Legende

Container

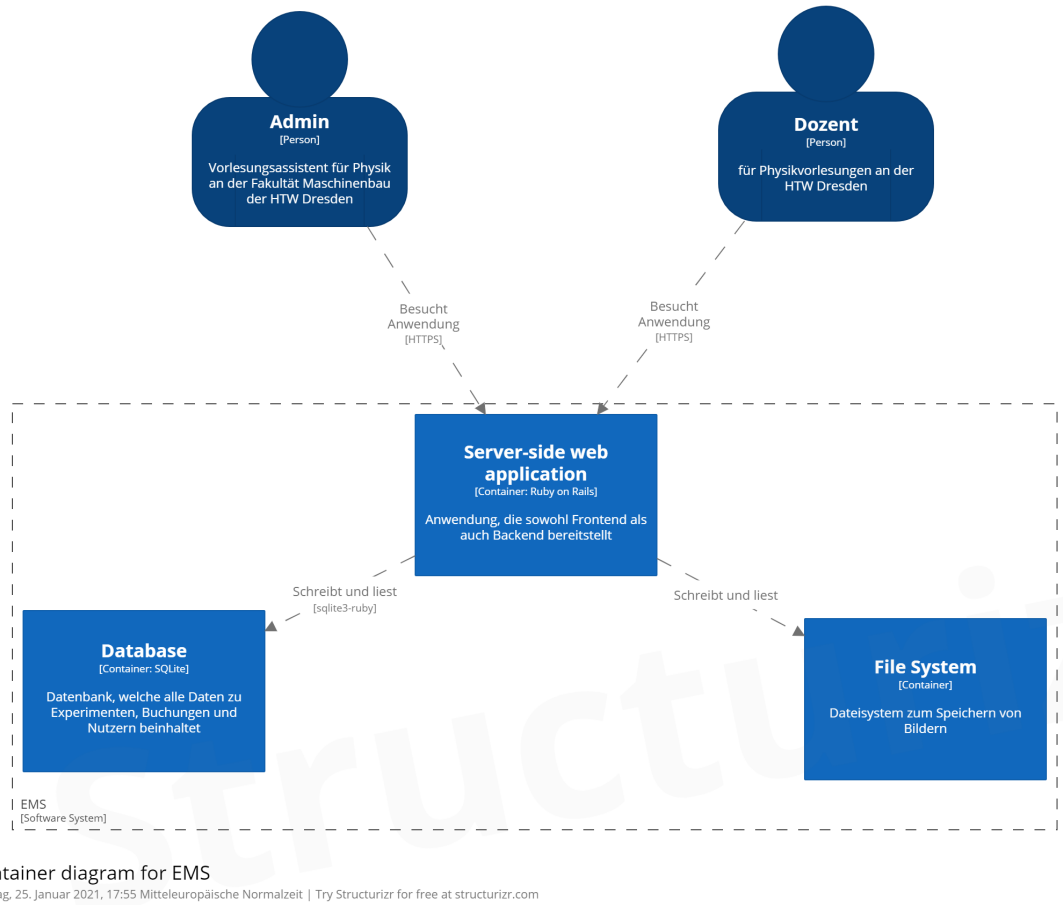


Abbildung 3. Container-Diagramm

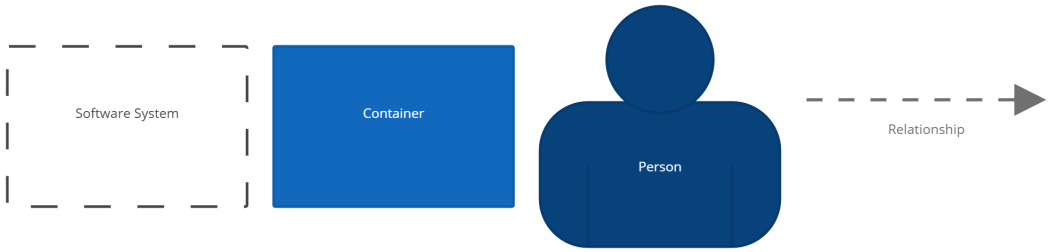
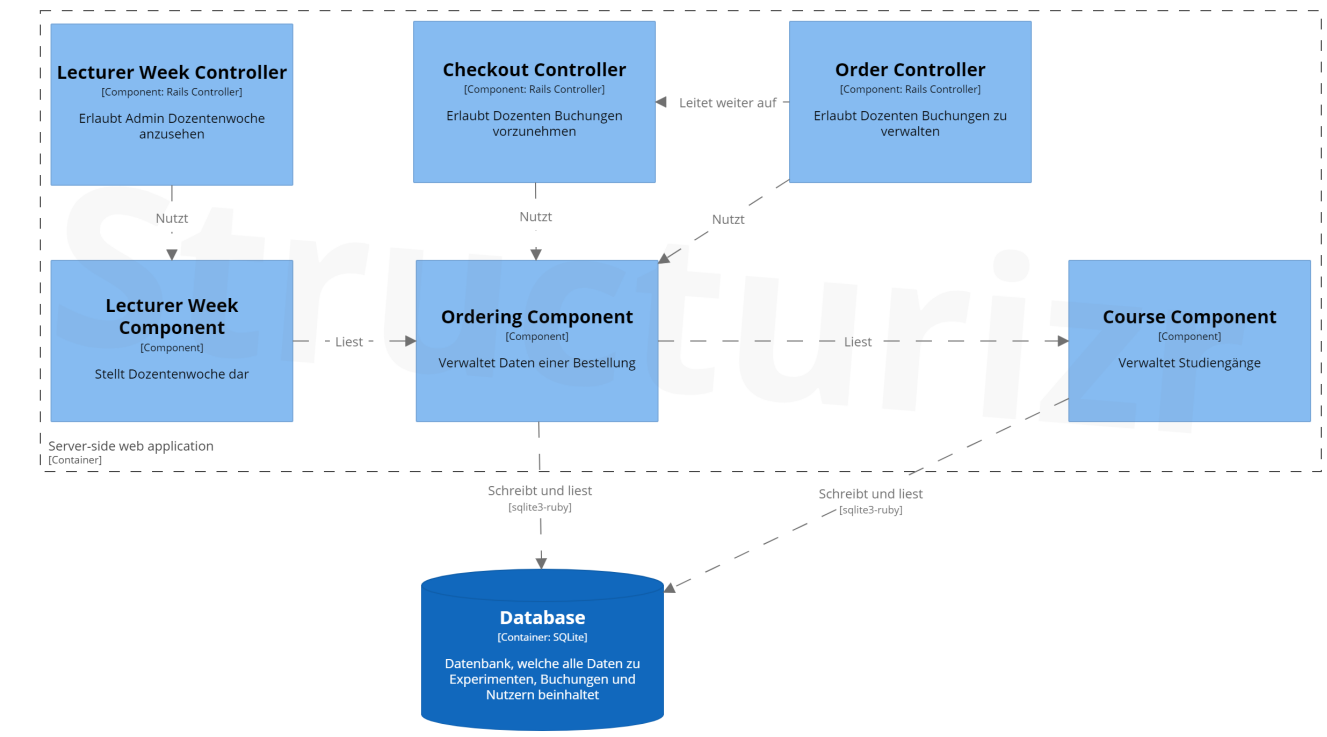


Abbildung 4. Container-Diagramm - Legende

Component



Component diagram for EMS - Server-side web application

Diagram created with Structurizr | Sonntag, 13. Juni 2021, 15:19 Mitteleuropäische Sommerzeit

Abbildung 5. Component-Diagramm

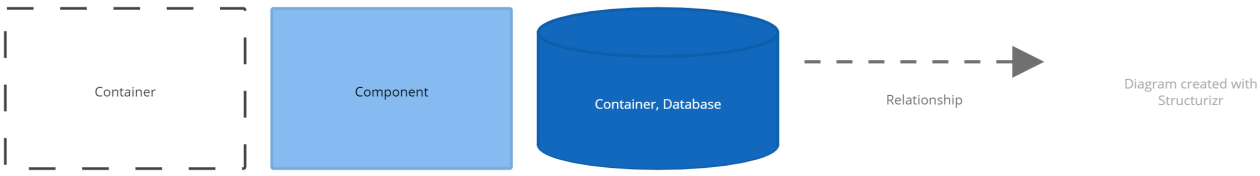


Abbildung 6. Component-Diagramm - Legende

Code

Zur Darstellung der Code-Ebene wurde ein UML-Klassendiagramm verwendet.

- hellblau gefärbte Packages dienen zur virtuellen Aufteilung der Klassen in ihre Komponenten (siehe Section 4.9.1.3, “Component”)
- lila gefärbte Attribute stellen sogenannte *virtuelle* Attribute dar, welche nicht in der Datenbank persistiert werden
- hellgrau gefärbte Klassen bilden Bibliotheksklassen ab

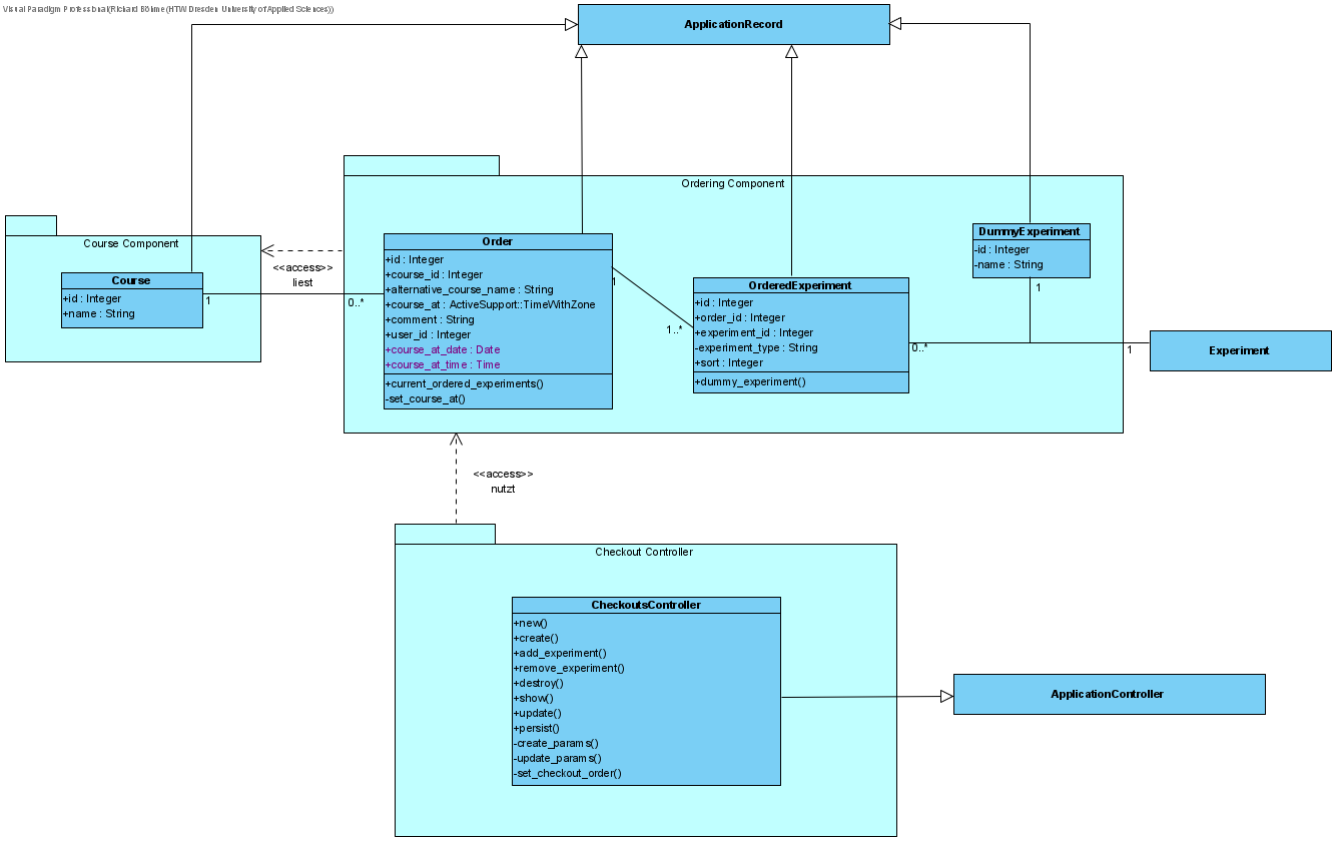


Abbildung 7. Klassendiagramm der Bestellkomponenten

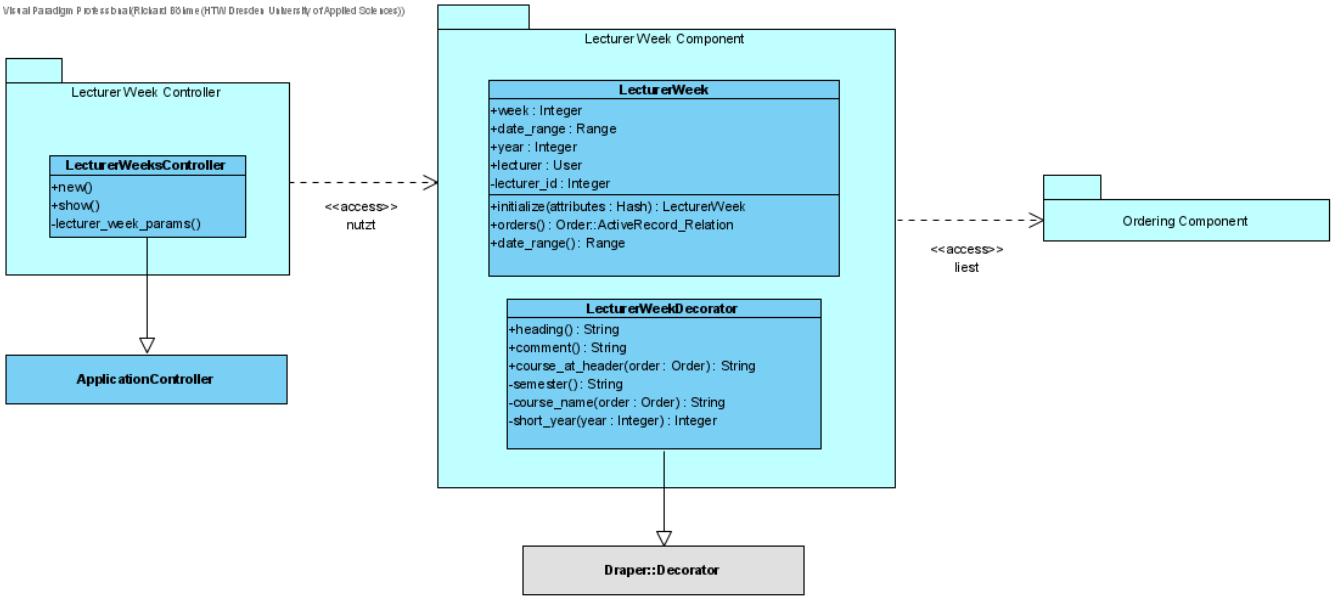


Abbildung 8. Klassendiagramm der Dozentenwochekomponenten

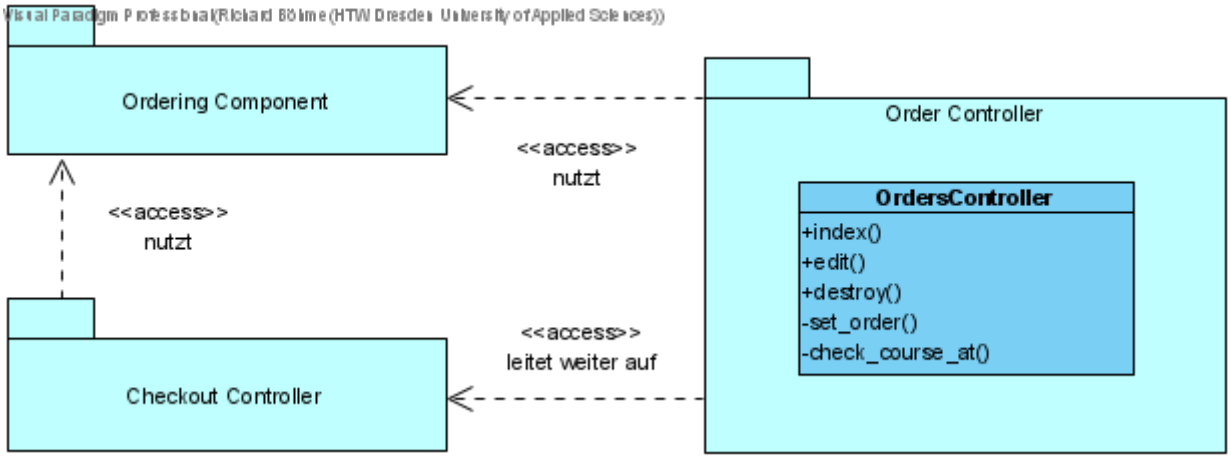


Abbildung 9. Klassendiagramm des Journals

4.9.2. Physische Sicht

Das System wird auf einer VM der Fakultät Maschinenbau betrieben.

Als Webserver wird NGINX verwendet.

Als Applikationsserver wird Phusion Passenger verwendet. Dieser skaliert die Anwendung automatisch mit mehreren Prozessen. Dabei werden maximal sechs Prozesse verwendet. Der genaue Skalierungsprozess ist in der Dokumentation von Passenger dokumentiert.

4.9.3. Use cases

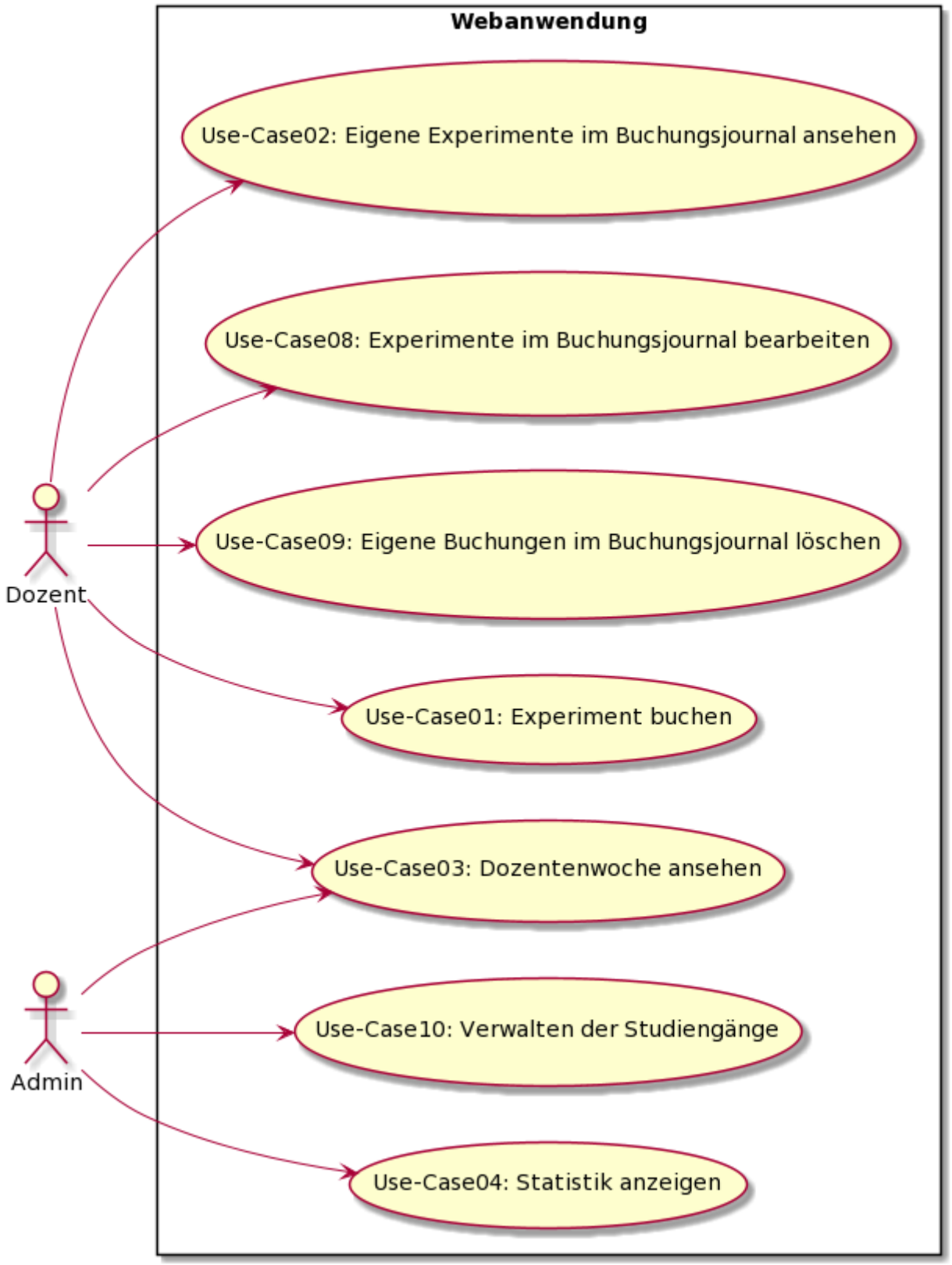


Abbildung 10. Use Case Diagramm

5. Softwaredokumentation

Der gesamte neu geschrieben Quellcode wurde nach den Konventionen von YARD dokumentiert. Die generierte Dokumentation liegt in `src/doc/index.html` vor. Alternativ kann auch der Dokumentationsserver von YARD gestartet werden:

```
gem install yard
cd src
yard server --reload
```

Dann kann die Dokumentation im Browser unter `http://localhost:8808` angesehen werden.

6. Codekonventionen

Zu Beginn der Weiterentwicklung wurde sich auf gewissene Codekonventionen geeinigt. Diese sind unter Section 8.1.3, “Konventionen” zu finden.

7. Deployment

7.1. Generelles

Das Deployment des EMS wurde von der bereits bestehenden Anwendung übernommen.

Zum Einsatz kommt die Bibliothek Capistrano. Die bereits bestehende Konfiguration der Bibliothek wurde weiter verwendet, allerdings wurden Änderungen vorgenommen um das Update der Ruby Version vornehmen zu können.

7.2. Testsystem

Um eine Weiterentwicklung vor der produktiven Nutzung durch alle Stakeholder testen zu können, wurde ein Klon des Produktivsystem als Testsystem eingesetzt.

7.3. Aktualisieren der Anwendung

7.3.1. Voraussetzungen

Der SSH-Public-Key der Person, die das Deployment durchführt, ist auf dem Zielsystem hinterlegt.

7.3.2. Auf einem System mit Ruby 3.0.1

Läuft die Anwendung auf dem System bereits mit der Ruby Version `3.0.1`, dann kann ein Deployment durch das Kommando `bundle exec cap <stage> deploy` durchgeführt werden. Mögliche Stages sind `production` und `test`.

Dabei wird der Stand installiert, der lokal vorliegt.

7.3.3. Auf einem System mit Ruby < 3.0.1

Folgende Schritte müssen durchgeführt werden um Ruby auf die verwendete Version zu aktualisieren. Alle Kommandos (außer Nr. 6) müssen auf der jeweiligen VM ausgeführt werden.

1. RVM (Ruby Version Manager) aktualieren

   ```
   rvm get stable
   ```

2. Neue Ruby Version aktualisieren

   ```
   rvm install 3.0.1
   ```

3. Neue `bundler` (Ruby Paketmanagement) Version installieren

```
gem install bundler
```

4. Yarn installieren (Javascript Paketmanagement)

```
curl -fsSL https://deb.nodesource.com/setup_16.x | sudo -E bash -
sudo apt-get install -y nodejs
sudo npm install --global yarn
```

5. Passenger (Anwendungsserver) aktualisieren

```
sudo apt-get --only-upgrade install passenger
```

6. Capistrano Deploy-Prozess anstoßen (Kommando lokal)

```
bundle exec cap <stage> deploy
```

   Hinweis: Es gilt dasselbe Verhalten wie oben beschrieben.

7. Neue Ruby Version als Standard einstellen

```
rvm use --default 3.0.1
```

7.4. Installation rückgängig machen

Eine Installation wird erst angewendet, wenn das Kommando `bundle exec cap <stage> deploy` erfolgreich war.

Soll eine installierte Version trotzdem rückgängig gemacht werden, muss das Kommando `cap <stage> deploy:rollback` verwendet werden. Alternativ kann einfach eine neue Version installiert werden. Mehr Informationen dazu sind in der Dokumentation von Capistrano zu finden.

7.5. Ordnerstruktur

Die Anwendung wird auf allen Systemen im Pfad `/var/www/app` installiert. Per Konvention von Capistrano befinden sich in diesem Verzeichnis folgende Ordner:

- **current**: Aktuell laufende Version. Dies ist ein Symlink auf einen Ordner im `releases` Ordner.
- **releases**: Enthält die letzten fünf installierten Versionen (inklusive der aktuell installierten Version)
- **shared**: Enthält Dateien und Ordner welche zwischen allen Releases geteilt werden (z.B. von Nutzern hochgeladene Dateien oder die Datenbank)

8. Anhang

8.1. A1: Ruby Einführung

8.1.1. Einführung - Was ist Ruby?

> "Ruby is a dynamic, open source programming language with a focus on **simplicity** and **productivity**. It has an elegant syntax that is natural to read and easy to write.
>
> — ruby-lang.org

Philosophie & Prinzipien

- Fokus auf "Developer Happiness"
- einfache Syntax: wer Englisch kann, kann auch Ruby Code lesen
- objektorientiert - *"Everything is an object"*
  - z.B. `true` ist eine Instanz der Klasse `TrueClass`

Eigenschaften

- wird zur Laufzeit interpretiert ⇒ kein Compiler notwendig
- dynamische Typisierung ⇒ keine feste Zuordnung zu Datentypen (siehe Variablen)
- automatische Speicherbereinigung

8.1.2. Language-Features - Wie nutzt man Ruby?

Variablen

dynamische Typisierung

Wie bereits erwähnt ist Ruby dynamisch typisiert. Deshalb ist z.B. folgender Code möglich und gewollt:

```
x = 5
# => 5
x = 'foo'
# => "foo"
```

Dies erlaubt eine sehr hohe Flexibilität, ist aber ggf. auch fehleranfällig.

nil

`nil` repräsentiert einen leeren Wert (ähnlich wie z.B. `NULL` Pointer in C). Jede Variable die nicht definiert ist, ist zunächst nil. (z.B. als Rückgabewert von Funktionen, Arrayzugriff auf Index welcher zu

groß ist,...)

```
x = []
x[1]
# => nil
```

Strings

String Interpolation

```
x = 5
"Max ist #{x} Jahre alt"
# => Max ist 5 Jahre alt
```

Verschiedene Varianten Strings zu erzeugen

Strings in Ruby können entweder mit einfachen oder doppelten Anführungszeichen definiert werden.

```
'foo' == "foo"
# => true
```

Der Unterschied liegt darin, dass man mit in einfachen Anführungszeichen definierten Strings keine Variablen einfügen kann.

```
x = 5
puts 'Max ist #{5} Jahre alt'
# => "Max ist #{5} Jahre alt"
```

Dafür ist diese Art Strings zu definieren etwas effizienter.

Die Ruby Community ist etwas gespalten, ob man standardmäßig alle Strings in einfachen oder doppelten Anführungszeichen schreiben sollte.

Symbole

Symbole sind ähnlich wie wiederverwendbare Strings. Sie sind **nicht** dafür da tatsächlichen Textinhalt zu symbolisieren und unterscheiden sich dahingehend stark von Strings. Ihr eigentlicher Nutzen ist ein Bezeichner für einen Wert darzustellen.

Ein Beispiel könnte eine Unterscheidung von verschiedenen Anwendungsfällen sein. Der Vorteil gegenüber Strings ist, dass nicht jedesmal ein neues Objekt im Speicher erstellt wird. Jedes gleiches Symbol ist das selbe Objekt im Speicher.

```ruby
x = :foo

# Symbol -> String
:foo.to_s
# => "foo"

# String -> Symbol
"foo".to_sym
# => :foo
```

Für ein Beispiel zum Nutzen siehe Section 8.1.2.3, “Bedingte Anweisungen”.

Booleans (Unterschied truthy, falsy)

Zusätzlich zu einfachen Booleans (`true`, `false`) wertet Ruby auch alle anderen Werte als *truthy* oder *falsy*. Dies erlaubt z.B. folgende if-Anweisungen:

```ruby
x = "foo"
if x
  puts x
end

x = nil
if x
  puts x
end

# => x würde nur einmal ausgegeben werden
```

Die einzigen Werte die als `falsy` gelten sind `false` und `nil`. Alle anderen Objekte sind `truthy`.

wichtige Datenstrukturen

Array

Arrays können einfach mittels `[]` initialisiert werden. Sie haben, anders als in anderen Sprachen, keine feste Größe und können beliebige Daten von verschiedenen Datentypen enthalten.

```ruby
array = [1, "foo", false]
# => [1, "foo", false]
```

Interessante Methoden:

```ruby
array.index("foo")
# => 1

array.map(&:to_s)
# => ["1", "foo", "false"]

array.each do |value|
  puts value
end
# => 1
#    "foo"
#    false

array.select { |value| value.is_a? Integer }
# => [1]

array << 'bar'
# => [1, "foo", false, "bar"]
```

Mehr Infos hier.

Hash

Hashes bilden einfache Wertepaare (Key/Value-Pairs). Typischweise sind die Keys Symbole oder Strings. Es können allerdings alle Werte zugeordnet werden. Es gibt dabei für Symbole eine extra Schreibweise, welche etwas eleganter ist. Zugriff auf den Wert erhält man über den jeweiligen Key.

```ruby
# für Symbole:
hash = { foo: "bar", bar: 1 }
hash[:bar]
# => 1

# generelle Syntax
hash = { "foo" => "bar", "bar" => 1 }
hash["bar"]
# => 1

hash["bar"] = 2
# => 2 (Hash: { "foo" => "bar", "bar" => 2 })
```

Interessante Methoden:

```ruby
hash.values
# => ["bar", 1]

hash.keys
# => ["foo", "bar"]

hash.except("foo")
# => { "bar" => 1 }

hash.slice("foo")
# => { "foo" => "bar" }

hash.each do |key, value|
  puts "Key: #{key}, Value: #{value}"
end
# => Key: foo, Value: bar
#    Key: bar, Value: 1
```

Mehr Infos hier.

Bedingte Anweisungen

Um bedingte Anweisungen zu schreiben braucht man in Ruby standardmäßig keine Klammern (weder geschweifte, noch runde).

```ruby
if x
# do something
elsif y
# do something else
else
# do something different
end
```

Für Verknüpfungen stehen `&&` für UND- und `||` für ODER-Verknüpfung zur Verfügung. Wobei `&&` als erstes ausgeführt wird, wenn keine Klammern verwendet werden.

If-Anweisungen geben immer den letzten Wert zurück. Wird die Bedingung nicht erfüllt wird `nil` zurückgegeben.

```ruby
x =
  if 2 == 1
    "bar"
  end
# => x = nil
```

unless

`unless` führt den Code genau dann aus, wenn die Bedingung nicht `truthy` ist bzw. wenn sie `falsy` ist.

```
x = true
unless x
  puts "x is false"
end
# => nothing is printed

# equal to:
if !x
  puts "x is false"
end
```

`unless` kann schwer zu verstehen sein, weshalb man es vor allem dann nicht nutzen sollte, wenn es einen `else` Block gibt.

if-Modifier

Einzeilige Operationen können durch eine Bedingung am Ende auf eine Zeile geschrieben werden. Auch hier sollte darauf geachtet werden, dass der Code übersichtlich bleibt.

```
some_method if x == 5

# equal to:
if x == 5
  some_method
end
```

Tenary Operator

Wie z.B. C hat auch Ruby einen Tenary-Operator:

```
age = 14
cost = (age > 18) ? 25 : 15
# (Klammern dienen nur zur Übersicht und können theoretisch weg gelassen werden)
```

Schleifen

Es gibt in Ruby zwar klassische for und while-Schleifen. Allerdings sieht man diese recht selten. Wenn man über eine Liste/Array iterieren will nutzt man die `each`-Methode. Zur Ausführung eines Codes x-mal nutzt man die `times`-Methode:

```ruby
x = ["foo", "bar", 1, 3]
x.each do |value|
  puts value
end
# => "foo"
#    "bar"
#    1
#    3

# Mit laufendem Index
x.each_with_index do |value, i|
  puts "i: #{i}, value: #{value}"
end
# => i: 0, value: foo
#    i: 1, value: bar
#    i: 2, value: 1
#    i: 3, value: 3

5.times do |n|
  puts n
end
# => 0
#    1
#    2
#    3
#    4

# oder:
for n in ["foo", "bar", 1, 3]
  puts n
end

while true do
  puts "infinite loop"
end
# equally to
loop do
  puts "infinite loop"
end
```

Methoden

Methoden werden in Ruby mit dem Keyword `def` eingeleitet. Danach folgt der Name der Methode und eine Argumentenliste. Aufgerufen werden Methoden mit ihrem Namen und einer Liste der zu übergebenden Argumente.

```ruby
def my_wonderful_method_name(x, y, z)
  puts "#{x}, #{y}, #{z}"
end

my_wonderful_method_name(1, 2, 3)
# => "1, 2, 3"
```

Hat die Methode keine Argumente, kann man die Klammern beim Funktionsaufruf und der Definition weglassen.

```ruby
def print_hello
  puts "Hello world!"
end

print_hello
# => "Hello world!"
```

Rückgabewerte

Der Rückgabewert ist immer das letzte Statement in einer Funktion.

```ruby
def print_and_return(x)
  puts x
  x
end

puts print_and_return(5)
# => 5
#    5
```

Es ist trotzdem möglich `return` zu nutzen um z.B. eine Funktion bedingt abzubrechen.

```ruby
def foo(x)
  return if x == 5
  return 4 if x == 4
  x / 2
end

foo(5)
# => nil

foo(4)
# => 4

foo(6)
# => 3
```

Standardargumente

Man kann Argumenten Standardwerte zuweisen (wie z.B. in C++)

```
def transform_name(name, action = :upcase)
  if action == :upcase
    name.upcase
  elsif action == :downcase
    name.downcase
  end
end

puts transform_name("Max")
# => "MAX"

puts transform_name("Max", :downcase)
# => "max"

puts transform_name("Max", :kebabcase)
# => nil
```

Keyword Argumente

Man kann Keyword Argumente nutzen, um Argumente hinzuzufügen, welche beim Aufruf mit dem Namen spezifiziert werden müssen.

```ruby
def foo(bar:)
  puts bar
end
foo(bar: 'test')
# => "test"
foo
# => missing keyword :bar

# Mit Standardargumenten:
def foo(bar: 'test')
  puts bar
end
foo
# => "test"

# Gemischt mit normalen Argumenten
# => Keyword Argumente müssen immer am Ende der Argumentliste stehen
def foo(x, bar: 2)
  puts x + bar
end
foo(2)
# => 4

foo(2, bar: 5)
# => 7
```

Splat()- und Double-Splat(*)-Operator

In Ruby ist die Argumentliste immer ein Array. Man kann ein Array mit dem Splat(*)-Operator zu einer Argumentliste konvertieren

```ruby
def foo(x, y, z)
  puts x, y, z
end

foo(*[1, "foo", false])
# => 1
#    "foo"
#    false
```

Der Splat Operator kann aber auch genutzt werden um die gesamte Argumentliste in einer Funktion zu repräsentieren.

```ruby
def foo(*args)
  puts args
end

foo(1, 2, 3)
# => 1
#    2
#    3

foo("bar", false, 3, 5, 7)
# => "bar"
#    false
#    3
#    5
#    7
```

Aus diesen Argumente kann man auch einzelne "herausziehen":

```ruby
def foo(a, *args)
  puts "special a: #{a}"
  puts "everything else: #{args.join(", ")}"
end

foo(1, "foo", "bar")
# => special a: 1
#    everything else: foo, bar
```

Dasselbe gilt für den Double-Splat Operator, Keyword Argumente und Hashes.

```ruby
def foo(a:, b: "test", **options)
  puts "a: #{a}"
  puts "b: #{b}"
  puts "options: #{options}"
end

foo(a: "foobar", c: false)
# => a: "foobar"
#    b: "test"
#    options: {c=>false}
```

Keine Sorge: Das kann schnell recht kompliziert werden und dient hier vorallem der Information.

Blocks, Procs & Lambdas

Ein spezielles Sprachfeature was aber sehr häufig genutzt wird sind Blocks, Procs und Lambdas. Einfach gesagt sind alle drei verschiedene Arten von "anonymen Funktionen". Die Bezeichnung ist nicht ganz korrekt und es gibt logischerweise Unterschiede zwischen den drei Arten, aber die sol-

len hier nicht weiter besprochen werden.

Blocks werden oft genutzt um Codestücke an bestimmte andere Funktionen weiterzureichen. Diese Funktionen können mit dem Keyword `yield` den übergebenen Block aufrufen. Dabei können auch Werte an den Block übergeben werden.

```
def foo
  puts "before yield"
  yield 3
  puts "after yield"
end

foo do |value|
  puts value * 2
end
# => before yield
#    6
#    after yield

# Alternative Syntax (vorallem besser für Einzeiler):
foo { |value| value * 2 }
```

Das ganze wird auch häufig in Funktionen verwendet die Ruby selbst liefert:

```
array = [1, 2, 3, 4, 5]
array.map { |value| value * 2 }
# => [2, 4, 6, 8, 10]
```

Wenn nur eine Methode genutzt wird gibt es noch einen schicken Shortcut:

```
array = ["foo", "bar"]
array.map(&:upcase)
# => ["FOO", "BAR"]
```

Mehr Informationen dazu gibt es z.B. hier.

Klassen

Da Ruby objektorientiert ist gibt es natürlich auch Klassen, Vererbung usw. Hier ein Beispiel für eine Klassenstruktur

```ruby
class Vehicle
end

class Train < Vehicle
end

class Car < Vehicle
end
```

Mittels `<` kann man also Vererbung abbilden.

Die Konstruktormethode heißt in Ruby `initialize` und ein Objekt einer Klasse kann mit `<Klasse>.new` erzeugt werden:

```ruby
class Vehicle
  def initialize(name, type)
    puts name, type
  end
end

class Car < Vehicle
  def initialize(name)
    super(name, "Car")
  end
end

Car.new("Audi")
# => Audi
#    Car
```

Mit `super` wird die Methode des Elternobjektes ausgeführt. Es kann dabei überall in der Methode stehen.

Instanzvariablen werden mit den Methoden `attr_reader` (für lesende Methode), `attr_writer` (für schreibende Methode) und `attr_accessor` (für beide) angelegt. Das bedeutet:

```ruby
class Vehicle
  attr_reader :type
  attr_accessor :name
  attr_writer :date
end

# erzeugt / ist gleich wie
class Vehicle
  def type
    @type
  end
  def name
    @name
  end
  def name=(value)
    @name = value
  end
  def date=(value)
    @date = value
  end
end
```

Die Variablen welche mit `@` beginnen sind dabei Instanzvariablen. So kann das obere Beispiel z.B. folgendermaßen umgeformt werden:

```ruby
class Vehicle
  attr_accessor :name, :type
  def initialize(name, type)
    @name = name
    @type = type
  end
end
v = Vehicle.new("Airbus", "Flugzeug")
v.name
# => "Airbus"
v.type
# => "Flugzeug"
```

Folgendermaßen definiert man Instanz- oder Klassenmethoden:

```ruby
class Vehicle
  attr_accessor :name, :type
  PRICES = { "Audi" => 120_000, "VW" => 50_000 } # hier eine Konstante
  def initialize(name, type)
    @name = name
    @type = type
  end

  # Klassenmethode
  def self.prices
    PRICES
  end

  # Instanzmethode
  def calculate_price
    PRICES[name]
  end
end

v = Vehicle.new("Audi", "Car")
v.calculate_price
# => 120000
v.name = "VW"
v.calculate_price
# => 50000

Vehicle.prices
# => {"Audi"=>120000, "VW"=>50000}
```

`self` referenziert immer das aktuelle Objekt.

Module

Module sind quasi "Funktionensammlungen", welche man zu Klassen hinzufügen kann. Man nutzt sie folgendermaßen:

```ruby
module Utils
  def sum(x, y)
    x + y
  end
end

class Vehicle
  include Utils

  attr_accessor :price

  def initialize(price)
    @price = price
  end

  def add_to_price(x)
    sum(x, price)
  end
end

v = Vehicle.new(3000)
v.add_to_price(2)
# => 3002
```

8.1.3. Konventionen

- Variablen, Methoden, Symbole, Dateinamen und Ordnernamen in Snakecase (`my_wonderful_variable`)
- Klassen in CapitalCase (`MyWonderfulClass`)
- Nutze keine for/while-Loops (stattdessen `[].each` oder `3.times`)
- Methoden die Booleans zurückgeben mit `?` enden lassen (z.B. `object.printable?`)
- Methoden die Objekte verändern mit `!` am Ende (Bang Methods)

```ruby
array = [1, 2, 3]
array.map(&:to_s)

array
[1, 2, 3]

array.map!(&:to_s)

array
["1", "2", "3"]
```

- Nutze Symbole statt Strings wenn möglich und sinnvoll (z.B. in Hashes)
- Zwei Leerzeichen als Einrückung

- Keine Leerzeichen o.Ä. (Whitespace) am Ende einer Zeile
- Styleguide: https://github.com/rubocop/ruby-style-guide

8.1.4. Dokumentation

Ein häufiges Tool zur Dokumentation von Ruby Code ist Yard. Es erlaubt die Dokumentation von Code über Kommentare sowie das Generieren von Entwicklerdokumentation aus diesen.

Beispiel:

```
# Reverses the contents of a String or IO object.           <----- Description
#
# @param [String, #read] contents the contents to reverse    <----- Argumente der
Funktion
# @return [String] the contents reversed lexically           <----- Returnwert
def reverse(contents)
  contents = contents.read if respond_to? :read
  contents.reverse
end
```

Dafür werden sogennante Tags genutzt (wie `@param` oder `@return`). Dokumentation zu den Tags findet man hier.

Integrationen finden sich für RubyMine (Ruby IDE von JetBrains) oder auch Visual Studio Code (Ruby Solargraph Extension).

8.1.5. Einrichtung

Generell ist es zu empfehlen Ruby in einem Unix-Betriebssystem zu entwickeln. Alternativ kann auch die Entwicklung mit dem Windows Subsystem erfolgen.

Ruby installieren geht am besten mit einem Versions-Manager. Ein Beispiel dafür ist RVM. So könnt ihr einfach Ruby Versionen installieren und zwischen ihnen wechseln.

```
gpg --keyserver hkp://pool.sks-keyservers.net --recv-keys
409B6B1796C275462A1703113804BB82D39DC0E3 7D2BAF1CF37B13E2069D6956105BD0E739499BDB

\curl -sSL https://get.rvm.io | bash -s stable --ruby
```

Hier geht es zu der Installationsanleitung für weitere Informationen.

Installierte Ruby Versionen ansehen:

```
rvm list
```

Ruby Version wechseln:

```
rvm use <version>
```

Ruby Version installieren:

```
rvm install <version>
```

Alternativen:

- Ruby Install
- asdf Version Manager mit dem Ruby Plugin
- rbenv
- chruby

8.1.6. Tipps und Tricks

Man kann jederzeit eine Ruby Konsole mit dem Command `irb` öffnen. Damit können dann alle möglichen Code-Snippets ausprobiert werden.

```
# Zeige Methoden eines Objektes an:
obj.methods.sort
# Entferne die des Standard-Objektes:
obj.methods.sort - Object.methods

# Erhalte die Klasse eines Objektes
obj.class
```

8.1.7. Hilfreiche Links

- https://rubyapi.org
- https://ruby-doc.org
- https://www.ruby-lang.org/en/
- https://github.com/ruby/ruby

8.2. A2: Rails Einführung

8.2.1. Einführung - Was ist Rails?

Rails im Gegensatz zu Ruby ist keine Programmiersprache sondern ein Framework oder eine Basis um eine Webandwendung in Ruby zu entwickeln. Es bringt verschiedene Funktionen mit sich um die Entwicklung einer Webanwendung zu beschleunigen und effizienter zu gestalten. So müssen kritische Stellen in der Applikation nicht selbst entworfen werden (das Rad nicht neu erfinden).

Prinzipien die Rails nutzt *(aus der Rails Doctrine)*:

- **Optimize for programmer happiness**: Ruby lässt grüßen
- **Convention over Configuration**: Entscheidungen die "unwichtig" sind werden direkt von Rails getroffen. Beispiel: Es braucht keine 3 Stunden Diskussion darüber wie der Primärschlüssel einer Tabelle heißen sollte.
- **The menu is omakase**: Rails gibt dir einen Stack an Frameworks und Bibliotheken mit denen du arbeiten kannst. Wie wenn man den Koch nach dem besten Gericht fragt.
- **No one paradigm**: Rails folgt nicht einem Paradigma. Es mischt teilweise welche, die sich eigentlich in Konflikt stehen. Wichtig ist es ein Paradigma an der Stelle anzuwenden wo es hin passt.
- **Exalt beautiful code**: Rails wurde entwickelt um "schönen" Code zu schreiben. Das geht natürlich mit dem ersten Punkt dieser Liste zusammen und hilft zusätzlich der Lesbarkeit.
- **Provide sharp knives**: Rails gibt Werkzeuge die falsch verwendet werden können. Ähnlich zu Ruby gibt es dem Entwickler viele Freiheiten, welche auch im falschen Sinne genutzt werden können. Nutzt man sie jedoch an der richtigen Stelle zum richtigen Zeitpunkt kann man umfangreiche Anforderungen einfach umsetzen oder kapseln.
- **Value integrated systems**: Rails liebt Monolithe - also eine Anwendung für ein ganzes System (entgegen zu Microservices). Es macht die Enwicklung einfacher. Natürlich braucht man manchmal mehrere Systeme oder Services aber oft reicht eine Anwendung um eine erfolgreiche Anwendung zu entwickeln.
- **Progress over stability**: Rails will sich weiter entwickeln was unter anderem dazu führt, dass große Änderungen notwendig werden um eine Anwendung auf eine aktuelle Version von Rails zu aktualisieren. Nur so jedoch erhofft man sich Fortschritt und Weiterentwicklung.
- **Push up a big tent**: Rails will möglich viele Leute in seiner Community sehen. Auch welche die nicht mit einigen Grundideen übereinstimmen. Diskussion bringt die Sprache Ruby und das Framework und somit auch die Community weiter.

8.2.2. Struktur

Die wichtigsten Verzeichnisse und Dateien einer Rails Applikation auf einen Blick:

- **app**: Hier befindet sich der eigentliche Applikationscode. (Models, Views, Controller, JS/CSS, ...).
- **config**: Hier finden sich alle möglichen Konfigurationen wie z.B. die Zugangsdaten zur Datenbank, die Routen, verschiedene Einstellungen für das Framework, Deploymenteinstellungen oder Ähnliches.
- **db**: Der db-Ordner beinhaltet Datenbank-Migrationen, das Schema der Datenbank sowie Testdaten für die Entwicklung.
- **lib**: Hier kann eigener Bibliothekscode abgelegt werden (selten verwendet).
- **public**: Alle Dateien die hier liegen sind statisch über die Webanwendung später erreichbar. (z.B. Favicon, Fehlerseiten, o.Ä.)
- **.ruby-version**: Enthält die Ruby Version der Anwendung (wird von Ruby Versionsmanagern genutzt um automatisch die Ruby Version zu wechseln).
- **Capfile**: Für das Deployment notwendig

- **Gemfile**: Enthält alle Abhängigkeiten zu Bibliotheken die benötigt werden

Es gibt noch ein paar mehr. Die oben genannten sind aber die wichtigsten und am meisten verwendeten.

8.2.3. Routen

Wird ein Request an die Anwendung gemacht wird zunächst geprüft welche Stelle im Code zuständig ist um die Anfrage zu bearbeiten. Das geschieht meistens in Abhängigkeit vom Pfad (www.example.com/**test/anmelden**) und der HTTP-Methode (Get, Post, Patch, Delete, ...). Diese Zuordnung geschieht in Rails über die Routen, welche in *config/routes.rb* definiert werden.

Rails stellt dafür verschiedene Methoden bereit. Weitere Informationen dazu können in den Rails Guides gefunden werden.

8.2.4. Model, View, Controller (MVC)

Controller

Eine Route, wie oben beschrieben, verweist dann auf eine Action in einem Controller. Ein Controller ist eine Ruby Klasse, welche in *app/controllers* liegt und von `ActionController::Base` erbt. Rails erstellt automatisch einen `ApplicationController`, von welchem dann meist weiter vererbt wird.

Eine Action ist nichts anderes als eine Methode. Diese wird also beim Zugriff über eine bestimmte Route aufgerufen und bestimmt was nun mit der Anfrage des Nutzers passieren soll.

Häufig werden HTML-Views generiert und zurückgegeben oder es wird auf einen anderen Pfad weitergeleitet. Rails hilft da natürlich. Weitergeleitet werden kann mittels der `redirect_to` Methode.

```
class PagesController < ApplicationController

  def test
    redirect_to '/'
  end

end
```

Oben stehendes Beispiel würde auf die Hauptseite weiterleiten.

Mittels der `render` Methode können Views gerendert werden (sowohl HTML als auch andere Formate).

```ruby
class PagesController < ApplicationController

  def test
    render 'show'
  end

end
```

Standardmäßig versucht Rails eine View zu rendern die in dem zum Controller zugehörigen Pfad liegt und den Namen der Action trägt. In dem Fall z.B. *app/views/pages/show* wenn kein `render` Befehl gemacht wird.

Die Kommunikation mit der View geschieht über Instanzvariablen (siehe Ruby Einführung). Diese können im Controller definiert werden und sind dann ebenfalls in der View verfügbar.

Innerhalb einer Action hat man zudem Zugriff auf einige wichtige Informationen die mit dem Request zusammenhängen. Zum einen kann man auf den Request selbst zugreifen mit der `request` -Methode. Viel wichtiger ist jedoch meistens die `params` Methode, welche alle Parameter zu einem Request mitbringt. Parameter sind in Rails zum einen zusätzliche Daten in dem Pfad wie `/experiments/1/show?foo=bar` und zum anderen Daten die über Formulare übermittelt werden. Es gibt ein Objekt der Klasse `ActionController::Parameters` zurück, welches in seinem Aufbau einem Hash ähnelt.

Warum ist es nun kein Hash? Es gibt die Möglichkeit Parameter direkt an eine Klasse zu übergeben, welche dann automatisch alle Attribute von diesen Parametern setzt. Nun wäre es aber möglich, dass die Parameter nicht das enthalten was man selbst erwartet (User Input). Ein klassisches Beispiel ist, dass ein Nutzer ein Attribut `admin` hat, welches ihn zum Admin macht. Natürlich wird man dies nicht einem Formular anbieten, aber Nutzer können Parameter auch händisch manipulieren. Um zu verhindern, dass solche Probleme auftreten, wurden Möglichkeiten geschaffen Parameter nach ihrem Namen zu erlauben. Häufig macht man das mit solch einem Ansatz:

```ruby
class UsersController < ApplicationController

  def update
    user = User.find(params[:id].to_i)
    user.update(user_params)
  end

  def user_params
    params.require(:user).permit(
      :name,
      :birth_date
    )
  end

end
```

Obiges Snippet würde nur Parameter in der Form `{ "user" ⇒ { "name" ⇒ "..", "birth_date" ⇒`

`"…" } }` erlauben.

Mehr Informationen zu Controllern sind in den Rails Guides zu finden.

Model

Models sind Klassen, welche in *app/models* liegen und von `ActiveRecord::Base` erben. Auch hier wird häufig eine Basisklasse `ApplicationRecord` verwendet.

Sie bilden eine Verbindung zwischen Datenbank und Klassen im Source Code. ActiveRecord ist die dazu gehörende Rails-eigene Bibliothek. Weitere Informationen zu detaillierter Nutzung dieser in Section 8.2.5, “ActiveRecord Basics”.

Models bilden also solche Klassen ab, die persistent gespeichert werden sollen. Sie besitzen Möglichkeiten zum Erstellen, Aktualisieren und Löschen von Einträgen ohne manuell irgendwelche SQL Queries schreiben zu müssen. Sie halten zudem Teile der Businesslogik die das jeweilige Objekt betrifft.

An der Stelle ein kurzer Einblick in ein typisches Dilemma in der Rails Welt: Kommt meine Businesslogik in den Controller oder in die Models? Dazu gibt es verschiedenste Meinungen und ich will hier keine von beiden bevorzugen oder sagen wie es am besten ist. Es ist nur eine Frage mit der sich Beginner oft herumtreiben und die eventuell zu schwer lesbaren oder unnötig komplexen Code führen kann. Als Daumenregel gilt für mich immer, dass man ein Mittelmaß finden sollte. Man entwickelt ein gewissen Gefühl dafür und auch professionelle Rails Entwickler haben damit immer mal Probleme (die Alternative zu dem Problem sind dann Service-Objekte).

Models können theoretisch auch einfache Ruby Klassen sein (die nicht von `ActiveRecord::Base`) erben. Das ist nur eben der Standard, da man häufig Objekte in einer Datenbank persistieren will. Alternativ kann man auch das Modul `ActiveModel::Model` inkludieren, welches die Persistenzschicht auslässt.

View

Views sind das was der Nutzer am Ende zu sehen bekommt. In einem Monolith ist das meistens HTML. Es könnte aber genauso gut JSON oder XML oder einfach nur Text sein.

Views leben in *app/views*. Standardmäßig legt man sie in einem zum Controller passenden Pfad ab (PagesController erwartet Views z.B. in *app/views/pages*).

Wie schon erwähnt kann man in Views auf Instanzvariablen aus Controllern zugreifen und diese nutzen um dynamische Inhalte zu generieren.

Slim

Standardmäßig werden in Rails standard-HTML Dateien verwendet. Die Möglichkeit in diesen Ruby Code zu nutzen nennt sich `erb` (Embedded Ruby). So hat man dann meistens `.html.erb` Dateien.

Rails unterstützt jedoch auch andere Sprachen um HTML zu generieren. Eine häufig genutzte Variante ist Slim. Die Kurzfassung der Syntax ist:

- HTML ohne schließende Tags

- Tags ohne <>-Zeichen
- Text durch | kennzeichnen
- Ruby-Ausdrücke mit einem Bindestrich davor
- Ruby-Ausdrücke dessen Ergebnisse angezeigt werden sollen mit einem Gleichheitszeichen davor

Beispiel:

```
<!-- app/views/users/show.html.erb -->
<h1><%= @user.name %></h1>

<p>Geburtsdatum: <%= @user.birth_date %></p>
```

```
/ app/views/users/show.slim
h1 = @user.name
p
  | Geburtsdatum:
  = @user.name
```

Hilfreiche Links:

- https://github.com/slim-template/slim
- https://html2slim.herokuapp.com

Layouts

Die meisten Views haben einen gemeinsamen Rahmen (HTML Head, usw.), welcher in Rails durch Layouts abgebildet wird. Diese "besonderen" Views liegen in *app/views/layouts* und sollten das Keyword `yield` enthalten an wessen Stelle die tatsächliche View beim Rendern eingefügt wird. Im Controller kann man dann u.A. mit der `layout` Methode wählen welches Layout genutzt werden soll.

Partials

Häufig möchte man Teile von Views wiederverwenden. Dafür legt man in Rails sogenannte Partials an. Die liegen an der selben Stelle wie andere Views. Allerdings muss der Name mit einem Unterstrich beginnen. So lässt sich z.B. einfach eine Liste darstellen:

```
/ app/views/users/_user.slim
li = user.name

/ app/views/users/index.slim
ul
  - @users.each do |user|
    = render partial: 'user', locals: { user: user }

    / oder etwas eleganter:
    = render 'user', user: user

  / oder noch etwas eleganter:
  = render partial: 'user', collection: @users

  / oder noch viel eleganter
  = render @users
```

Man sieht an dem Beispiel es gibt viele Wege zum Ziel. Partials müssen auch nicht unbedingt mit einem bestimmten Objekt zusammenhängen. Eine anderes Beispiel wo Partials nützlich sind ist ein Footer oder ein Werbebanner.

Helpers

Manchmal benötigt man etwas komplexeren Code um Sachen in einer View darstellen zu können. In Standard Rails gibt es dafür Helper-Klassen. Diese liegen unter *app/helpers* und können in allen Views verwendet werden. Im Controller sind die Helper mit der `helpers`-Methode verfügbar.

Form Helpers

Um das bauen von Formularen zu vereinfachen, liefert Rails u.A. den Form Helper `form_with`.

Viele Rails Projekte nutzen allerdings ein Gem, welches das ganze noch weiter vereinfachen soll. Das Gem heißt SimpleForm.

Angenommen man hat also ein Model `Person` mit den Attributen `name` und `age` und will diese in einem Formular erfassen. So kann man SimpleForm folgendermaßen nutzen:

```
/ app/views/people/_form.slim

= simple_form_for @person do |f|
  = f.input :name
  = f.input :age, as: :integer, input_html: { min: 0 }
  = f.submit
```

So hätte man ein einfaches Formular um einen Nutzer anzulegen oder zu bearbeiten. Mehr Infos findet sich in der Dokumentation von SimpleForm.

Assets

Alle CSS-, Javascript sowie Bilddateien heißen Assets. Sie sind im Ordner `app/assets` zu finden und werden von Rails kompiliert und genutzt.

Statt CSS wird die Variante Sass genutzt die zu CSS kompiliert wird. Sass bietet eine einfachere Syntax und wird daher gern genutzt.

Neue Sass Dateien können mittels `@import "file"` zu einer Datei hinzugeführt werden.

Neue JS Dateien können mittels `//= require file` zu einer Datei hinzugefügt werden.

In dem Layout einer Anwendung nutzt man dann einen Rails Helper um die verschiedenen Dateien (meistens eine JS, eine CSS und x Bilddateien) in das HTML einzubetten.

```
stylesheet_link_tag 'filename'
javascript_include_tag 'filename'

image_tag 'filename'
```

In neuen Anwendung nutzt man häufig, das in der Javascript-Welt beliebte Build-Tool Webpack um Assets zu verwalten.

8.2.5. ActiveRecord Basics

ActiveRecord stellt die Verbindung zwischen Objekten im Code und der Datenbank dar. Jede Klasse stellt (im Normalfall) eine Tabelle dar und jedes Objekt der Klasse eine Zeile in dieser Tabelle. Dafür bietet ActiveRecord verschiedene Methoden um Objekte anzulegen, zu aktualisieren oder zu löschen.

create

Mit der `create` Methode lässt sich ein Objekt in der Datenbank anlegen. Die Instanz des Objektes wird zurückgegeben.

```
person = Person.create(name: 'Max')
# => INSERT INTO people (name) VALUES ('Max')
```

Hinweis: In ActiveRecord werden meist numerische IDs als Primärschlüssel genutzt. Das von `create` zurückgegebene Objekt hat dann eine ID. Man muss sich also nicht um die ID Vergabe selbst kümmern.

Hinweis: Häufig werden in Rails Models mit Timestamps angelegt (`created_at`, `updated_at`). Diese werden ebenfalls automatisch aktualisiert.

persisted?

Die `persisted?`-Methode gibt an ob ein Model bereits in der Datenbank gespeichert wurde. Dies kann z.B. genutzt werden um zu Prüfen ob ein `create` geklappt hat.

update

Mit der `update`-Methode kann ein Eintrag in der Datenbank aktualisiert werden. Es gibt true/false zurück je nach dem ob das Aktualisieren geklappt hat.

```
person.update(name: 'Max 2')
# => UPDATE people SET name = 'Max 2' WHERE id = ? (person.id)
```

save

Zusätzlich zum direkten Aktualisieren oder Anlegen von Datenbankzeilen kann auch das Objekt selbst zunächst angelegt/aktualisiert werden ohne die Datenbank zu aktualisieren. Die Methode gibt true/false zurück je nach dem ob die Operation in der Datenbank erfolgreich war.

```
person = Person.new
person.name = 'Max'
person.save
# => INSERT ...
person.name = 'Max2'
person.save
# => UPDATE ...
```

destroy

Um ein Objekt aus der Datenbank zu löschen gibt es die `destroy`-Methode. Diese gibt ebenfalls zurück ob die Operation erfolgreich war.

```
person.destroy
# => DELETE FROM people WHERE id = ? (person.id)
```

Hinweis: Es gibt auch eine `delete` Methode. Diese macht ähnliches, führt jedoch keine Callbacks (Section 8.2.5.11, “Callbacks”) aus und führt nur die eine SQL Query aus.

Associations

Ein typisches Datenmodell sieht Beziehung zwischen seinen Klassen vor. Auf Datenbankebene wird dies, je nach Beziehungstyp, durch Fremdschlüssel abgebildet. Rails vereinfacht die Arbeit mit diesen Beziehungen im Code. Dafür müssen sie in den Models definiert werden.

Fremdschlüssel haben standardmäßig die Form `model_id`. Wobei `model` der Name der Assoziation ist. Die Klasse leitet Rails von dem Assoziationsnamen ab. Sowohl den Fremdschlüsselname als auch die referenzierte Klasse kann aber auch unabhängig vom Assoziationsnamen gewählt werden. Dies passiert dann über Optionen an jeweiligen Assoziationsmethoden.

1:1

Für 1:1 Beziehungen stellt Rails die Methoden `belongs_to` und `has_one` bereit. Der Unterschied liegt

darin, wo der Fremdschlüssel definiert ist.

Beispiel:

```
# app/models/person.rb
class Person < ApplicationRecord
  has_one :profile
  # Prüft das Profile einen Fremdschlüssel person_id hat.
end

# app/models/profile.rb
class Profile < ApplicationRecord
  belongs_to :person
  # Prüft Fremdschlüssel person_id.
end
```

Dadurch werden folgende Abfragen möglich:

```
person = Person.create
profile = Profile.create(person: person)
person == profile.person
# => true
profile == person.reload.profile
# => true
```

1:n

Für 1:n Beziehungen stellt Rails die Methoden `has_many` und `belongs_to` bereit.

Beispiel:

```
# app/models/house.rb
class House < ApplicationRecord
  has_many :windows
  # => Rails leitet aus windows die Klasse Window ab (von Plural zu Singular da
has_many)
end

# app/models/window.rb
class Window < ApplicationRecord
  belongs_to :house
  # => Rails leitet aus house die Klasse House ab
end
```

Wie man es von der klassischen Datenbankmodellierung kennt, besitzt `Window` dann den Fremdschlüssel zu `House` (standardmäßig `house_id`).

Dadurch werden folgende Abfragen möglich:

```
house = House.create
window1 = Window.create(house: house)
window2 = Window.create(house: house)

house = house.reload
house.windows.size == 2
# => true
house.windows == [window1, window2]
# => true
window1.house == house
# => true
```

m:n

Für m:n Beziehungen stellt Rails die Methode `has_many_and_belongs_to` bereit. Mehr Informationen dazu gibt es in den Docs.

where

Um Abfragen an der Datenbank durchzuführen gibt es unter anderem die `where`-Klassenmethode. Mit ihr kann man ein WHERE in SQL abbilden.

Beispiele:

Standardfall:

```
people = Person.where(lastname: 'Mustermann')
# => SELECT * FROM people WHERE lastname = 'Mustermann'
```

Mehrere Attribute:

```
people = Person.where(lastname: 'Mustermann', age: 18)
# => SELECT * FROM people WHERE lastname = 'Mustermann' AND age = 18
```

Mehrere Werte:

```
people = Person.where(age: [16, 18, 20])
# => SELECT * FROM people WHERE age IN (16, 18, 20)
```

Eigene Bedingung:

```
min_age = 18
people = Person.where('age >= ?', min_age)
# => SELECT * FROM people WHERE age >= 18
```

Nach Assoziationen:

```
windows = Window.where(house: House.first)
# => SELECT * FROM windows WHERE house_id = 1
```

Negiert:

```
people = Person.where.not(age: 18)
# => SELECT * FROM people WHERE age != 18
```

find und find_by

Oft brauch man nur ein Ergebnis einer WHERE Abfrage. Hierfür nutzt man `find` bzw. `find_by`. Mit `find` kann man nur nach der ID suchen:

```
Person.find(1)
```

Achtung: Findet `find` kein Ergebnis wirft es eine `ActiveRecord::NotFound` Exception. Wird diese in einem Controller geworfen, rendert Rails automatisch eine 404-Seite.

Mit `find_by` sucht man nach einer eigenene Bedingung so wie mit `where`:

```
Person.find_by(firstname: 'Max', lastname: 'Mustermann')
```

Man sollte darauf achten `find_by` nur zu nutzen, wenn man sicher ist, dass es nur ein Ergebnis gibt. Andernfalls bekommt man einfach das Erste was die Datenbank ausgibt zurück. Findet `find_by` kein Objekt wird `nil` zurückgegeben.

order

ORDER BY SQL Anweisungen bildet Rails mit der `order`-Methode ab. Diese kann z.B. mit der `where` Methode kombiniert werden.

```
people = Person.where('age > 18').order(:age)
# => SELECT * FROM people WHERE age > 18 ORDER BY age

people = Person.order(age: :desc)
# => SELECT * FROM people ORDER BY age DESC
```

Preloading

Häufig hat man folgendes Ablauf im Code:

```ruby
# app/controllers/houses_controller.rb
class HousesController < ApplicationController
  def show
    @house = House.find(params[:id])
  end
end
```

```ruby
/ app/views/houses/show.slim
h1 = @house.name
ul
  = @house.windows.each do |window|
    li = window.name
```

Dies würde dazu führen, dass jedes `Window` Objekt einzeln im SQL abgerufen wird. Man spricht dann von einer sogenannten N+1 Query, wobei N die Anzahl an Fenstern wäre und die eine Query wäre die um das Haus zu finden.

Auf Datenbankebene gibt es dafür Joins. Rails bietet verschiedene Möglichkeiten an um diese durchzuführen. Um obiges Problem zu lösen, müsste man nur den Controller-Code anpassen:

```ruby
# app/controllers/houses_controller.rb
class HousesController < ApplicationController
  def show
    @house = House.find(params[:id]).includes(:windows)
  end
end
```

Mehr dazu gibt es in den Docs.

Callbacks

Callbacks sind Methoden die zu einem bestimmten Zeitpunkt automatisch ausgeführt werden. So zum Beispiel vor oder nach dem Speichern eines Objektes.

Es ist möglich einen Callback als Block direkt zu definieren oder einen Methodennamen zu übergeben, der dann aufgerufen wird.

```
class Person < ApplicationRecord
  before_update do
    puts "before update"
  end

  after_update :log_update

  def log_update
    puts "after update"
  end
end
```

Eine Liste aller Callbacks und weitere Informationen, wie z.B. das bedingte Ausführen von Callbacks, sind hier dokumentiert.

Scopes

Scopes sind eine Möglichkeit häufig verwendete Queries an einem Model zu definieren. Sie können dann als Klassenmethoden des Models aufgerufen werden.

Beispiel:

```
class Person < ApplicationRecord
  scope :adult, ->() { where('age >= 18') }
end

adults = Person.adult
# => SELECT * FROM people WHERE age >= 18
```

Es gibt auch die Möglichkeit Argumente an Scopes zu übergeben:

```
class Person < ApplicationRecord
  scope :older_than, ->(age) { where('age > ?', age) }
end

people = Person.older_than(17)
# => SELECT * FROM people WHERE age > 17
```

Nested Attributes

Häufig hat man den Fall, dass man ein Eltern-Model mit mehreren Assoziation in einem Formular darstellen will. Damit man nicht jedes Model selbst anlegen muss, bietet Rails die Methode `accepts_nested_attributes_for :relation` an.

Diese legt dann die Methode `relation_attributes=` am Model an. Somit ist es möglich einen Parameter-Hash der Form

```
{
  relation_attributes: {
    foo: 'bar'
  }
}
```

direkt an den Konstruktor des Models zu übergeben. Rails kümmert sich dann um das Anlegen, Löschen oder Bearbeiten der Relation.

Es ist auch möglich mehrere Assoziationen an die Methode zu übergeben:

```
class Person < ApplicationRecord
  belongs_to :profile
  has_many :parents, class_name: 'Person'

  accepts_nested_attributes_for :profile, :parents
end
```

Die wichtigste Option der Methode ist `allow_destroy`, mit welcher es auch möglich wird mittels eines Parameters (`_destroy`) eine Assoziation aufzulösen und das jeweilge Model zu löschen.

8.2.6. ActiveModel Basics

Hier wird nur auf ActiveModel in Verbindung mit Rails eingegangen.

Validations

Nicht jeder Wert den ein Nutzer in einem Formular eingibt ist auch semantisch korrekt. Um die Eingaben an einem Model zu prüfen gibt es eine ganze Menge an Validierungen die Rails bereitstellt. Dabei gibt es immer einen gleichen Aufbau um Validierungen zu definieren.

```
class Person < ApplicationRecord
  validates :age, numericality: true
end
```

Das obige Beispiel validiert z.B. dass das Attribut `age` eine Zahl sein muss.

Rails bietet standardmäßig bereits viele Validierungen an, die hier nicht alle aufgelistet werden sollen. Eine Liste findet man hier.

Viele Validierungen erlauben auch zusätzliche Optionen. Diese gibt man dann folgendermaßen an:

```
class Person < ApplicationRecord
  validates :age, numericality: { greater_than_or_equal_to: 18 }
end
```

Bedingte Validierungen kann man folgendermaßen definieren:

```
class Person < ApplicationRecord
  validates :age, numericality: true, if: ->{ self.orders.alcoholic.present? }
end
```

Man kann auch mehrere Validierungen kombinieren:

```
class Person < ApplicationRecord
  validates :age,
      presence: true,
      numericality: true
end
```

Man kann auch Methoden definieren die zum Aufruf der Validierungen ausgeführt werden:

```
class Person < ApplicationRecord
  validate :age_restrictions

  def age_restriction
    if age < 18
      errors.add(:age, :invalid)
    end
  end
end
```

Validierungen werden automatisch, beim Speichern eines Models aufgerufen (z.B. über `save` oder `create`). Man kann sie auch manuell mit der Methode `valid?` aufrufen.

Ist das Model nicht valide, geben die Methoden `false` zurück.

Errors

Jedes Model hat die Methode `errors`. Wenn Validierungen für das Model fehlschlagen, kann man damit dem Nutzer Fehlermeldungen anzeigen lassen. Die Methode liefert ein Objekt der Klasse `ActiveModel::Errors` zurück, welches wiederum verschiedene Methoden besitzt. Die wichtigstens Methoden um über Errors zu iterieren sind `full_messages` (Array der Fehlermeldungen) und `messages` (Hash mit dem Attribut als Key und den Nachrichten als Value) (z.B. für Fehlermeldungen).

Man kann selbst eigene Errors an das Error-Objekt hinzufügen (z.B. in einer einenen Validierungs-Methode).

```
errors.add(:attribute, :type, message: '')
```

Wenn man die Option `message` weglässt, wird versucht den Fehler mit I18n zu übersetzen.

8.2.7. Migrations

Um Änderungen am Datenbankschema vorzunehmen nutzt man in Rails Migrationen. Das sind Dateien die eine Änderung am Schema beschreiben (Tabelle anlegen, Feld hinzufügen, Feld ändern, usw.). Migrationen sollten umkehrbar sein. Alle Migrationen liegen im Ordner `db/migrate`.

Rails hält das aktuelle Datenschema in der Datei `db/schema.rb`. Dabei findet sich auch die aktuelle Schemaversion die genutzt wird um herauszufinden, ob neue Migrationen ausgeführt werden müssen.

Eine neue Migration kann man mit dem Befehl `rails g migration <name>` hinzufügen.

Mehr Informationen dazu was man mit Migrationen machen kann gibt es hier.

8.2.8. I18n

I18n ist eine Bibliothek zum Übersetzen von Texten in einer Anwendung. Man wählt einen Schlüssel, nach welchem die Anwendung dann in einer `.yml` Datei sucht und den entsprechenden Text anzeigt.

Mehr dazu kann hier nachgelesen werden.

8.2.9. Einfaches CRUD

Ein einfaches CRUD (Create, Read, Update, Delete) lässt sich in Rails innerhalb kürzester Zeit implementieren:

Neue App anlegen:

```
rails new blog
cd blog
```

Mittels eines Generators können vorgefertigte Dateien angelegt werden:

```
rails generate scaffold Post name:string content:text author:string
```

Dieser Command wird folgende Dateien anlegen und befüllen:

- `app/models/post.rb`
- `app/controllers/posts_controller.rb`
- `app/views/posts/index.html.erb`
- `app/views/posts/edit.html.erb`
- `app/views/posts/show.html.erb`
- `app/views/posts/new.html.erb`
- `app/views/posts/_form.html.erb`

- `db/migrate/0000000_create_posts.rb`
- ... und einige mehr

Migrationen ausführen:

```
rails db:migrate
```

Den Server starten:

```
rails s
```

Und zu `localhost:3000/posts` navigieren!

8.2.10. Best Practices

- Klassennamen müssen zu Ihrem Pfad passen. (z.B. `app/models/api/player.rb` muss die Klasse `Api::Player` definieren)
- Halte dich an den Ruby Styleguide
- Füge nicht unnötig Gems zum Projekt hinzu
- Nutze **nur** gesicherte Parameter
- Baue keine eigenen SQL Queries mit Nutzereingaben (ohne ActiveRecord zu nutzen)
- Teste deinen Code (sowohl Unit- als auch Systemtests)

8.2.11. Interessante Commands

```
# Öffnet Rails Konsole => irb im Rails Context
rails c

# Beispieldaten (Seeds) laden
rails db:seed

# Routen anzeigen
rails routes
```

Debugging im Code ist möglich mittels

```
binding.pry
```

8.2.12. Einrichtung

- Ruby installieren

Neues Projekt:

- `gem install rails`
- `rails new` - um ein neues Projekt anzulegen

Bestehendes Projekt:

```
# Node JS installieren
apt install nodejs libsqlite3-dev

# Projekt klonen (falls noch nicht getan)
git clone https://github.com/Schmiddl99/experimenteverwaltung-i2.git

# Verzeichnis betreten
cd experimenteverwaltung-i2/src

# Aktuellen Branch verwenden (andere sind noch kaputt)
git checkout 196_richard_ruby-and-rails-update

# Bundler installieren (falls noch nicht getan)
gem install bundler

# Gems installieren
bundle install

# Datenbank anlegen / Migrationen vornehmen
rails db:prepare

# Server starten
rails s
```

8.2.13. Hilfreiche Links

- https://api.rubyonrails.org
- https://guides.rubyonrails.org
- https://github.com/rails/rails