

MODUL 133

TEIL 4: MVC: MODEL, CON- TROLLER. LIST-, SHOW-, NEW-, CREATE-, EDIT-, UP- DATE-, DELETE-METHODEN IN RAILS REALISIEREN

Ralph Maurer / Michael Abplanalp

Inhaltsverzeichnis AB133-04

Modul 133: Webapplikationen mit Session-Handling realisieren

Teil 4: MVC: Model, Controller. list-, show-, new-, create-, edit-, update-, delete-Methoden in Rails realisieren

Rails Model.....	2
Beziehungen zwischen Models	3
Validierungen im Model.....	4
Rails Migrations	5
Tabellen erstellen	5
Eine Rails Migration erstellen	8
Models "Book" und "Subject" testen.....	9
Rails Controller	10
Lokale Variablen und Instanzvariablen	12
Books-Controller	12
Subjects-Controller	15
Praktische Arbeit	16
Aufgabe 1	16
Aufgabe 2	16
Aufgabe 3	17





Rails Model

In AB133-03 haben wir das Model `user` erstellt. Es war eine einzige Tabelle ohne Beziehungen. Zur Vervollständigung soll noch der Umgang mit mehreren Tabellen, Beziehungen und Validierungen im Model erklärt werden.

Angenommen, wir erzeugen zwei Models:

1. Fachgebiet
2. Buch

Die zwei Models stehen wie folgt in Beziehung: Fachgebiet \leftrightarrow Buch = (1:m)

Wir sprechen auch von einer one-to-many Beziehung. Da wir englische Bezeichner nutzen wollen, übersetzen wir Fachgebiet in `Subject` und Buch in `Book` und erstellen diese Models. Der Name des Models sollte jeweils mit einem Grossbuchstaben beginnen und in Einzahl stehen:

```
$ rails g model Subject
$ rails g model Book
```

Beachten Sie, dass wir keine Attribute und Datentypen im Kommando `rails generate model` mitgeben. Ziel ist es, diese zu programmieren, so dass die Applikation ohne Neuerstellung der einzelnen Models von Hand z.B. auf einen anderen Server migriert werden kann.

Für das Model `Subject` wird folgender Code bzw. werden folgende Dateien generiert:

```
> invoke active_record
> create db/migrate/20180430133029_create_subjects.rb
> create app/models/subject.rb
> invoke test_unit
> create test/models/subject_test.rb
> create test/fixtures/subjects.yml
```

Für das Model `book` wird folgender Code bzw. werden folgende Dateien generiert:

```
> invoke active_record
> create db/migrate/20180430133034_create_books.rb
> create app/models/book.rb
> invoke test_unit
> create test/models/book_test.rb
> create test/fixtures/books.yml
```

Mit den Befehlen

```
$ rails g model Subject
$ rails g model Book
```

passieren hauptsächlich zwei Dinge: Einerseits wird der Rails-Generator angewiesen, Models für `Subject` und `Book` zu erstellen, damit später Instanzen abgespeichert werden können (ein Fachgebiet bzw. Buch).

Andererseits wird für die Models `Subject` und `Book` je eine Migrationsdatei generiert mit einer Grundstruktur für das Erstellen der Tabellen. Diese Dateien werden ergänzt mit den Tabellendefinitionen, der Code ist reines Ruby, wir müssen also keine SQL-Statements kennen. Mit anderen Worten: **Die Tabellen werden mit `rails g model` noch nicht erstellt!**

Wir werden diese Migrationsdateien im nächsten Kapitel (*Rails Migrations*) näher betrachten.

Beachten Sie, dass Sie Model-Bezeichner immer in Singularform verwenden. Dies ist ein Rails-Paradigma, dem Sie bei jedem Erstellen eines Models folgen sollten. Denn die Beziehungen verlangen je nach Fall die Pluralform. Z.B. bedeutet die 1:m-Beziehung `subject:book`, dass ein Fachgebiet mehrere Bücher haben kann, weshalb `book` zu `books` wird.

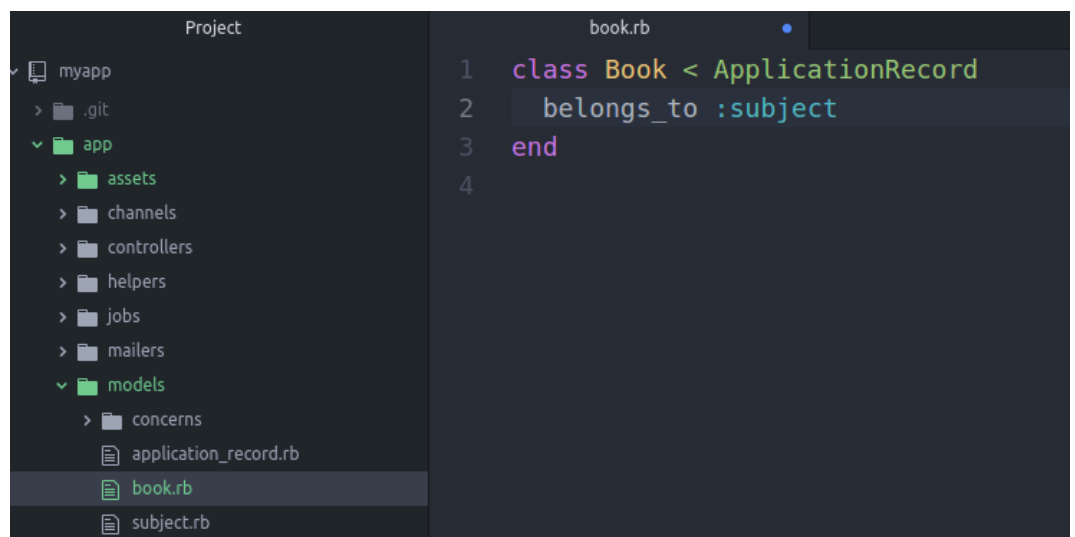
Für uns sind die Dateien `app/models/subject.rb` und `app/models/book.rb` wichtig: In diesen Dateien können wir Beziehungen und Validierungen abbilden.

Beziehungen zwischen Models

Rails kennt 6 Typen von Beziehungen zwischen Models:

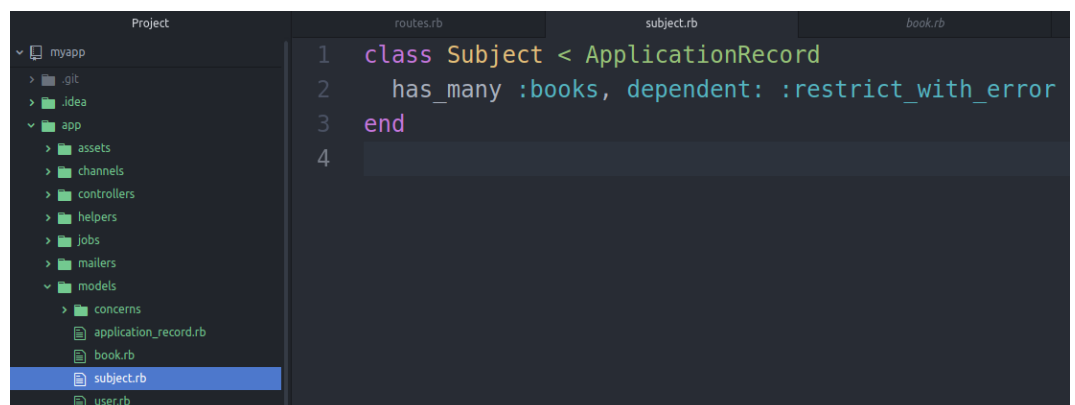
- › `belongs_to` (1:1 oder m:1-Beziehung)
- › `has_one` (1:1-Beziehung)
- › `has_many` (1:m-Beziehung)
- › `has_many :through` (n:m-Beziehung mit Zwischentabelle, die ein Ereignis darstellt)
- › `has_one :through` (1:1-Beziehung mit Zwischentabelle)
- › `has_and_belongs_to_many` (n:m-Beziehung mit Zwischentabelle)

Definieren wir zunächst die einfache Beziehung `book:subject` (ein Buch wird genau einem Fachgebiet zugeordnet). Hierzu öffnen wir die Datei `app/models/book.rb` und erweitern die Klasse `Book` mit der Beziehung `belongs_to`:



`belongs_to` bedeutet, dass im Model `Book` ein Fremdschlüssel erstellt wird, der in Beziehung zum Primärschlüssel des Models `Subject` steht.

Im Model `Subject` definieren wir, ob ein Fachgebiet mit nur einem oder mehreren Büchern beschrieben werden kann. Zudem legen das Verhalten beim Löschen eines Fachgebietes fest:



`has_many` bedeutet, dass ein Fachgebiet mehrere Bücher haben kann. Hier sehen wir, dass `books` im Plural geschrieben wurde. Nun wird auch klar, warum wir für die Models englische Begriffe in der Einzahl verwenden: Für die Mehrzahl wird einfach ein `s` angehängt.

Mit `dependent`: definieren wir, was passiert, wenn ein Fachgebiet gelöscht wird. `:restrict_with_error` bedeutet, dass ein Fachgebiet nur gelöscht werden kann, wenn keine Bü-



Falls Sie sich im Details über die sechs Beziehungstypen und deren Unterschiede befassen wollen, empfiehlt sich der *RailsGuides*, Kapitel *Active Record Associations*:
http://guides.rubyonrails.org/association_basics.html

Das Kapitel ist auch im Ordner `03_Arbeitsblaetter` als PDF-Datei mit dem Namen `AB133-03_Beziehungen.pdf` verfügbar.

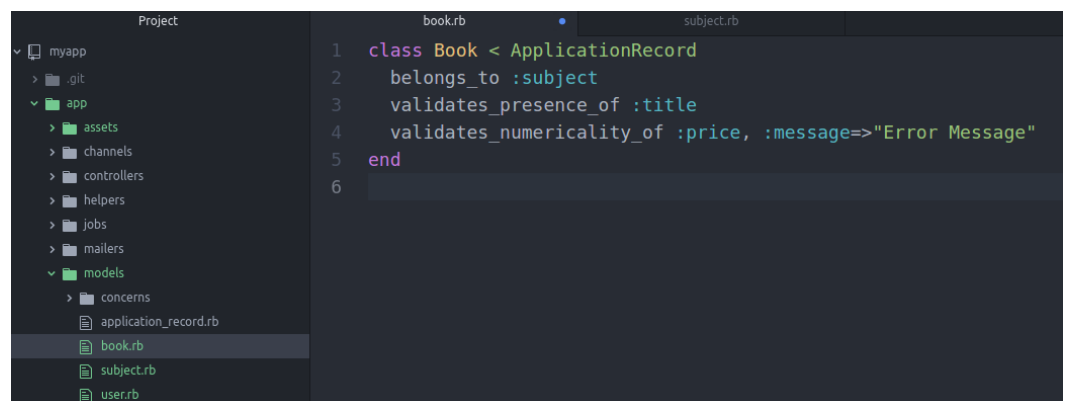
cher eine Referenz darauf haben. Ansonsten wird ein Fehler generiert.

Validierungen im Model

Das Rails-Model verwaltet direkt Validierungen der einzelnen Model-Attribute. Wir wählen für das Model `Book` zwei Bedingungen:

- › `validates_presence_of` (bedeutet NOT NULL, d.h. kann nicht leer sein).
- › `validates_numericality_of` (nur numerische Werte sind möglich).

Wir wenden die zwei Validierungen auf die noch nicht existierenden Attribute `title` und `price` an:



```
Project
└─ myapp
   └─ app
      ├── assets
      ├── channels
      ├── controllers
      ├── helpers
      ├── jobs
      ├── mailers
      └─ models
         ├── concerns
         ├── application_record.rb
         └─ book.rb
            ├── subject.rb
            └─ user.rb
```

```
1 class Book < ApplicationRecord
2   belongs_to :subject
3   validates_presence_of :title
4   validates_numericality_of :price, :message=>"Error Message"
5 end
6
```



Validierungen von Modelattributen werden im *RailsGuides* im Kapitel *Active Record Validations* beschrieben: http://guides.rubyonrails.org/active_record_validations.html

Das Kapitel ist auch im Ordner `03_Arbeitsblaetter` als PDF-Datei mit dem Namen `AB133-03_ValidierungModel.pdf` verfügbar.



Rails Migrations

Migrations sind eine praktische Möglichkeit, Datenbanken strukturiert und organisiert zu erstellen und zu ändern. *Active Record* hält fest, welche Migrationen bereits durchgeführt worden sind. Deshalb können Sie einfach den Code im Model ändern und `rails db:migrate` ausführen. *Active Record* ermittelt, welche Migrationen durchgeführt werden müssen.

Migrationen werden in Ruby geschrieben, sie sind unabhängig von der verwendeten Datenbank. Und Migrationen werden versioniert, d.h. man kann jederzeit zu einer früheren Version zurückkehren.

Was alles mit Rails Migrations möglich ist:

- › `create_table(name, options)`
- › `drop_table(name)`
- › `rename_table(old_name, new_name)`
- › `add_column(table_name, column_name, type, options)`
- › `rename_column(table_name, column_name, new_column_name)`
- › `change_column(table_name, column_name, type, options)`
- › `remove_column(table_name, column_name)`
- › `add_index(table_name, column_name, index_type)`
- › `remove_index(table_name, column_name)`

Migrationen unterstützen alle gängigen Datentypen:

- › `string`
- › `text`
- › `integer`
- › `float`
- › `datetime` und `timestamp`
- › `date` und `time`
- › `binary`
- › `boolean`

Folgende Validierungen können auf Attribute angewandt werden:

- › `limit (:limit => "50")`
- › `default (:default => "blah")`
- › `null (:null => false, implies NOT NULL)`

Tabellen erstellen

Beim Erstellen der beiden Models wurden folgende Migrationsdateien erstellt:

- › `myapp/db/migrate/<Nummer>_create_books.rb`
- › `myapp/db/migrate/<Nummer>_create_subjects.rb`

Der Inhalt von `<Nummer>_create_books.rb` sieht wie folgt aus:

```
20200124093913_create_books.rb
1  class CreateBooks < ActiveRecord::Migration[5.2]
2    def change
3      create_table :books do |t|
4
5        t.timestamps
6      end
7    end
8  end
```

Die Methode `change` wird ausgeführt, wenn die Änderungen der Migration auf die Datenbank angewendet werden. Mit `create_table` wird die Tabelle `books` erstellt. Das kleine `t` nach `do` ist ein Shortcut für die Tabelle `books`, damit wir mit die Attribute mit Datentyp erstellen können.

Mit `t.timestamps` werden die beiden Attribute `created_at` und `updated_at` vom Typ Datum erstellt. Erstellungsdatum `created_at` und Änderungsdatum `updated_at` werden automatisch von Rails nachgeführt und sollten deshalb nicht geändert, sondern nur gelesen werden.

Ergänzen Sie die Datei `<Nummer>_create_books.rb` wie folgt:

```
1  class CreateBooks < ActiveRecord::Migration[5.2]
2    def change
3      create_table :books do |t|
4        t.string :title
5        t.float :price
6        t.integer :subject_id
7        t.timestamps
8      end
9    end
10 end
```

`t.string`, `t.float`, `t.integer` sind die Datentypen der Attribute. `:title`, `:price`, `:subject_id` sind die Namen der Attribute.

Für alle Attribute könnten wir Eigenschaften festlegen wie `limit: 32` (max. Länge von Attributwerten = 32) oder `null: false` (keine leeren Attributwerte möglich). Es ist jedoch sinnvoller, diese Eigenschaften im Model zu definieren. Denn das Model kann jederzeit (auch zur Laufzeit) geändert werden, ohne dass die Tabellenstruktur angepasst werden müsste.

Wir haben weiter oben im Book-Model festgelegt: `validates_presence_of :title`, was der Attributeigenschaft NOT NULL entspricht.

Ergänzen Sie die Datei `<Nummer>_create_subjects.rb` wie folgt:

```
1 class CreateSubjects < ActiveRecord::Migration[5.2]
2   def change
3     create_table :subjects do |t|
4       t.string :name
5       t.timestamps
6     end
7
8     Subject.create name: "Rails"
9     Subject.create name: "Linux"
10    Subject.create name: "SQL"
11    Subject.create name: "Python"
12    Subject.create name: "Bash"
13  end
14 end
```

In der Methode `change` erstellen wir die Tabelle `subjects` und füllen mit `Subject.create` gleich fünf Datensätze in die Tabelle ein: *Rails*, *Linux*, *SQL*, *Python* und *Bash*.

Mit dem Ausführen der Rails-Migration werden die Tabellen generiert und im Falle der Fachgebiete werden gleich fünf Datensätze erstellt:

```
$ rails db:migrate
```



Falls beim Ausführen des obigen Kommandos ziemlich viel Text ausgegeben wird, gibt es wahrscheinlich Fehler in Ihrem Code. Scrollen Sie nach oben und prüfen Sie die Fehlermeldungen. Möglicherweise wird zu Beginn folgendes angezeigt:
StandardError: An error has occurred, this and all later migrations canceled:

Bei erfolgreicher Migration sollte in etwa folgendes angezeigt werden:

```
== 20180702141834 Books: migrating =====
-- create_table(:books)
-> 0.0018s
== 20180702141834 Books: migrated (0.0020s) =====

== 20180702141838 Subjects: migrating =====
-- create_table(:subjects)
-> 0.0049s
== 20180702141838 Subjects: migrated (0.0333s) =====
```

- ● ● Rails verwendet standardmässig für jede Applikation eine SQLite-Datenbank. Diese finden Sie unter `myapp/db/development.sqlite3`. Falls Sie eine Rails-Migration wiederholen müssen, weil es Fehler in den Migrationsdateien hat: Es kann sein, dass einige Tabellen angelegt worden sind, andere aber nicht. Mit der `sqlite3`-Konsole können Sie die Datenbank öffnen und Tabellen löschen, falls notwendig.

Sie können bzw. dürfen keine Anpassungen an der Datenstruktur direkt in der Datenbank vornehmen. Denn die Rails-Applikation würde in einem solchen Fall nicht mehr korrekt funktionieren!

Eine Rails Migration erstellen

Wenn wir zu einem späteren Zeitpunkt eine Tabellenstruktur anpassen wollen oder wenn wir z.B. Daten in eine Tabelle importieren wollen, können wir eine Rails Migration erstellen.

Das Rails-Kommando zum Erstellen von Migrationen sieht – analog zum Controller und Model – wie folgt aus:

```
$ rails generate migration <Name>
```

Rails Kommando Parameter «erzeugen» Typ «migration» Aussagekräftiger Name der Migration

Wir stellen fest, dass wir ein Attribut in der Tabelle `books` vergessen haben: Wir möchten eine Beschreibung für die Bücher hinzufügen (Attribut `description` mit dem Datentyp `string`).

Erstellen Sie folgende Migration, um das neue Attribut der Tabelle `books` hinzuzufügen:

```
$ rails generate migration AddDescriptionToBooks
```

Ergänzen Sie die Datei `db/migrate/<Nummer>_add_description_to_books.rb` wie folgt:

```
1 class AddDescriptionToBooks < ActiveRecord::Migration[5.2]
2   def change
3     add_column :books, :description, :string
4   end
end
```

`add_column` fügt das Attribut `description` vom Datentyp `string` der Tabelle `books` hinzu.

Dazu müssen Sie die Änderungen wieder in die Datenbank schreiben:

```
$ rails db:migrate
```

Models "Book" und "Subject" testen

Ein Model können wir erst testen, nachdem die Migration auf die Datenbank angewendet worden ist, also wenn die Tabellen erstellt worden sind.

Für das Testen eignet sich insbesondere die Rails-Konsole:

```
$ rails c
```

Verbindung zum Model etablieren:

```
2.4.2 :001 > Book.connection
```

Struktur der Tabelle `books` anzeigen:

```
2.4.2 :002 > Book
=> Book(id: integer, title: string, price: float, subject_id:
integer, description: text, created_at: datetime)
```

Datensätze aus der Tabelle `subjects` anzeigen.

```
2.4.2 :012 > Subject.find(1,2,3,4,5)
Subject Load (0.3ms)  SELECT "subjects".* FROM "subjects" WHERE
"subjects"."id" IN (1, 2, 3, 4, 5)
=> [#<Subject id: 1, name: "Rails">, #<Subject id: 2, name:
"Linux">, #<Subject id: 3, name: "SQL">, #<Subject id: 4, name:
"Python">, #<Subject id: 5, name: "Bash">]
```



Rails Controller

Wir haben schon viel über Controller erfahren und auch Controller angewendet. Ein Controller ist die logische Einheit (mit der Programmlogik) zwischen Benutzer, View und Model. Der Controller ist ein Dienstleister und bietet wichtige Funktionen an.

Den Rails Controller-Generator haben wir bereits für die Erstellung statischer Seiten in AB133-02 kennengelernt:

```
$ rails generate controller <verzeichnis> <seite1> <seite2> ...
```

Diagramm zur Erklärung der Parameter:

- `rails`: Rails Kommando
- `generate`: Parameter «erzeugen»
- `controller`: Typ «controller»
- `<verzeichnis>`: Verzeichnis mit statischen Seiten
- `<seite1> <seite2> ...`: Liste der Seiten im Verzeichnis

In diesem Beispiel wenden wir ihn ohne die Parameter Verzeichnis und Seite(n) an:

```
$ rails generate controller <controller-Name>
```

Diagramm zur Erklärung der Parameter:

- `rails`: Rails Kommando
- `generate`: Parameter «erzeugen»
- `controller`: Typ «controller»
- `<controller-Name>`: Name des controllers

Der Befehl zum Erstellen der Controller `Books` und `Subjects` lautet wie folgt:

```
$ rails generate controller Books  
$ rails generate controller Subjects
```

Achten Sie darauf, den ersten Buchstaben des Controller-Namens immer gross und den Namen in Mehrzahl zu schreiben!

Es werden die Dateien `myapp/app/controllers/books_controller.rb` und `myapp/app/controllers/subjects_controller.rb` erstellt.

Der Inhalt der Books-Controller sieht wie folgt aus:

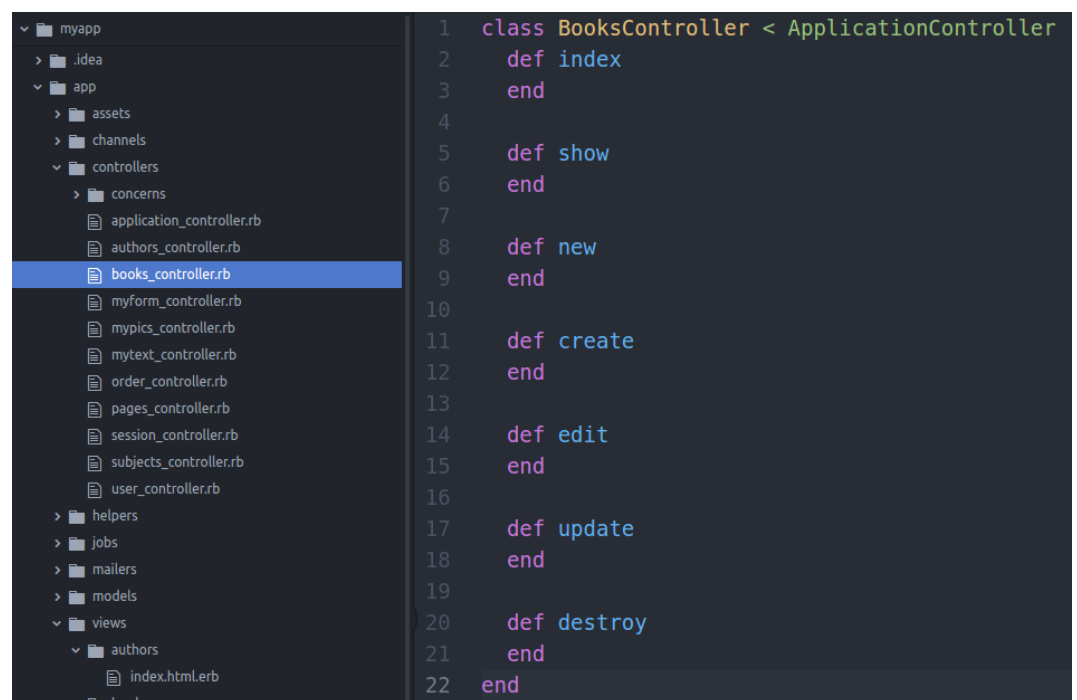
```
1 class BooksController < ApplicationController  
2 end  
3
```

Analog sieht auch der Subjects-Controller aus.

Wir ergänzen die Klasse `BooksController` mit allen notwendigen Methoden, damit wir später die Book-Objekte bearbeiten und anzeigen können:

Methode	Beschreibung
<code>index</code>	zeige alle erfassten Bücher an
<code>show</code>	zeige die Details eines Buches (Objekt <code>book</code>)
<code>new</code>	zeige das Formular zum Erfassen eines neuen Buches an
<code>create</code>	erstelle ein neues Objekt <code>book</code> und schreibe es in die Datenbank
<code>edit</code>	zeige das Formular zum Ändern eines Buches an
<code>update</code>	schreibe Änderungen am Objekt <code>book</code> in die Datenbank
<code>destroy</code>	lösche ein Objekt <code>book</code>

Erstellen Sie die Methodendeklarationen in der Klasse `BooksController`:



Da wir die Fachgebiete direkt in der Konsole erfasst haben, ergänzen wir die Klasse `SubjectsController` vorläufig nur mit einer Methode:

Methode	Beschreibung
<code>index</code>	zeige alle erfassten Fachgebiete an
<code>show</code>	zeige alle Bücher eines bestimmten Fachgebietes an

Wir betrachten im Folgenden alle Methoden einzeln. Der Code wird in den Dateien `myapp/app/controllers/book_controller.rb` und `myapp/app/controllers/subjects_controller.rb` erfasst.

Lokale Variablen und Instanzvariablen

Lokale Variablen sind nur innerhalb einer Klasse gültig und können nicht überall verwendet werden.

Wir benötigen aber Variablen, die sowohl im Controller als auch in der View nutzbar sind. Deshalb speichern wir Daten aus dem Model in Instanzvariablen. Instanzvariablen werden zum Speichern von Objekten verwendet.

Man unterscheidet lokale Variablen und Instanzvariablen durch die Schreibweise. Instanzvariablen beginnen immer mit dem `@`-Zeichen, lokale Variablen nicht.

Books-Controller

Die folgenden Methoden werden im Books-Controller erfasst, Datei `app/controller/books_controller.rb`.

index-Methode

Die `index`-Methode soll alle erfassten Bücher ausgeben. Hierzu können wir die Methode `all` aufrufen: `Book.all`.

Der Rückgabewert ist ein Array mit allen im Model angelegten Büchern. Damit das Array mit allen Büchern in einer View angezeigt werden kann, müssen wir es in eine Instanzvariable speichern. Wir nennen die Variable `@books`.

Die `index`-Methode sieht wie folgt aus:

```
def index
  @books = Book.all
end
```

Die `index`-Methode wird später in einer View beantragt. In der View werden wir uns um die Darstellung der Daten kümmern und Links zu weiteren Views bereitstellen. Die URL zur View soll wie folgt aussehen: <http://localhost:3000/books>

show-Methode

Die `show`-Methode soll ein Buch mit allen Attributen retournieren:

```
def show
  @book = Book.find(params[:id])
end
```

Die Zeile `@book = Book.find(params[:id])` beauftragt Rails, ein Buch zu finden mit der `id` (Primärschlüssel), die vom entsprechenden Link von der Webseite stammt. `params[:id]` ist ein formaler Parameter der Methode `find`.

Das `params`-Objekt erlaubt die Parameterübergabe zwischen unterschiedlichen Methodenaufrufen. Angenommen, die gesamte Bücherliste wird in der View <http://localhost:3000/books> angezeigt, so kann man ein bestimmtes Buch anklicken und der Primärschlüssel des Buches wird mit dem `params`-Objekt an die `show`-Methode weitergeleitet.

Das Buch wird geöffnet und in der dafür vorgesehen View angezeigt:

<http://localhost:3000/books/show/:id>

new-Methode

Die `new`-Methode weist Rails an, eine Seite mit Namen `new` anzuzeigen. Wir wollen ja ein Formular erstellen, um ein neues Buch zu erfassen. Wegen der 1:n-Beziehung müssen wir eine Liste mit Fachgebieten zur Verfügung stellen, damit das Buch einem Fachgebiet zugeordnet werden kann:

```
def new
  @book = Book.new
  @subjects = Subject.all
end
```

Die Zeile `@book = Book.new` erstellt ein Book-Objekt in der Instanzvariablen `@book`. Damit können wir einerseits im Formular-Helper auf dieses (leere) Objekt zugreifen. Andererseits wird bei einem Fehler nicht wieder ein leeres Formular geladen, sondern die bereits gemachten Eingaben werden angezeigt.

Die Zeile `@subjects = Subject.all` stellt in der Instanzvariablen `@subjects` alle erfassten Fachbereiche als Array zu Verfügung:

Die `new`-Methode soll von der View <http://localhost:3000/books/new> aufgerufen werden. Es wird ein Formular angezeigt, in dem die Attribute für das Buch erfasst werden können.

create-Methode

Sobald die Daten mit dem HTML-Formular erfasst worden sind, sollen diese an die `create`-Methode übermittelt und im Model gespeichert werden. Diese Methode hat keine eigene View, sondern zeigt bei Erfolg die `index`-View und bei Fehler die `new`-View an:

```
def create
  @book = Book.new(book_params)

  if @book.save
    redirect_to action: "index"
  else
    @subjects = Subject.all
    render action: "new"
  end
end

def book_params
  params.require(:book).permit(:title, :price, :subject_id, :description)
end
```

`Book.new(book_params)` weist Rails an, ein neues Buch-Objekt zu erstellen und in der lokalen Variablen `book` zu speichern. Der Parameter `book_params` ist eine Hilfsmethode, deren Aufgabe es ist, alle Eigenschaften des neuen Objekts zu sammeln. Die Eigenschaften werden dann der `create` Methode mittels `param`-Objekt übergeben.

Danach folgt eine `if`-Bedingung, die den Benutzer zur `index`-Methode weiterleitet, falls das Buch erfolgreich im Model abgespeichert werden konnte. `redirect_to` ist ein Befehl, mit dem man innerhalb des Controllers auf andere Methoden oder auch auf andere Seiten springen kann. In diesem Fall führt die Methode automatisch zur View `index`.

Sollte die Speicherung nicht erfolgreich sein, wird der Benutzer zurück zur `new`-Methode geleitet (Methode `render`). `render` wird angewendet, um direkt eine View anzuzeigen, ohne vorher Methoden des Controllers aufzurufen. Wenn also die fehlerhaften Daten wieder im Formular angezeigt werden sollen, müssen sie vorher in Instanzvariablen gespeichert werden.

Aus diesem Grund ist auch `@subjects = Subject.all` nötig: `render` zeigt <http://localhost:3000/books/new> an und es sollen wiederum alle erfassten Fachbereiche aufgelistet werden.

`params.require(:book).permit(:title, :price, :subject_id, :description)` bedeutet, dass die Methode `require` die Instanz des Buches in das `params`-Objekt legt und die Eigenschaften `title`, `price`, `subject_id` und `description` erlaubt sind (`permit`).



Wir verwenden in der `create`-Methode die Instanzvariable `@book` und nicht die lokale Variable `book`. Das Speichern würde zwar mit `book` auch funktionieren. Wenn jedoch ein Fehler auftritt und `render action: "new"` ausgeführt wird, dann bleiben die Daten im Formular nur erhalten, wenn wir die Instanzvariable `@book` einsetzen.

edit-Methode

Die `edit`-Methode ist fast identisch mit der `show`-Methode: Beide Methoden zeigen anhand der `id` ein Objekt in einer View an. Der einzige Unterschied besteht darin, dass die `show`-Methode ein Objekt nur anzeigt und die `edit`-Methode eine Aktion zum Ändern eines Objekts auslöst. Der Code sieht wie folgt aus:

```
def edit
  @book = Book.find(params[:id])
  @subjects = Subject.all
end
```

Diese Methode zeigt anhand des Primärschlüssels ein ausgewähltes Buch-Objekt an, das man in der View mit einem HTML-Formular verändern kann. Die Fachgebiete (`subjects`) müssen mitgegeben werden, damit auch das Fachgebiet eines Buches verändert werden kann.

Die `edit`-Methode wird via <http://localhost:3000/books/edit/:id> aufgerufen.

Das Speichern in die Datenbank wird ähnlich wie bei der `new`- und `create`-Methode in der eigenständigen Methode `update` realisiert.

update-Methode

Die `update`-Methode wird nach der `edit`-Methode ausgeführt, sobald der Benutzer die Daten geändert und auf *Speichern* geklickt hat. Alle Änderungen werden im Model gespeichert.

Die `update`-Methode sieht wie folgt aus:

```
def update
  @book = Book.find(params[:id])

  if @book.update_attributes(book_params)
    redirect_to action: "index"
  else
    @subjects = Subject.all
    render action: "edit"
  end
end
```



```
end
```

In Analogie zur `create`-Methode werden alle Attribute im Model gespeichert bzw. aktualisiert. Deshalb benutzen wir `book.update_attributes(book_params)` und nicht `book.save(book_params)`. `save` erstellt einen neuen Datensatz, `update_attributes` führt die Aktualisierung eines bestehenden Objekts durch.

Der restliche Code entspricht der `create`-Methode (siehe oben). Wir verwenden wieder die Methode `book_params`, die wir bereits oben definiert haben.

Auch die `update`-Methode hat keine eigene View, wir leiten bei Erfolg an die `index`-Methode bzw. bei Fehler an die `edit`-Methode weiter.

destroy-Methode

Die `destroy`-Methode dient dazu, einen Datensatz zu löschen. Diese soll von der Index-View via Link <http://localhost:3000/books/destroy/:id> aufrufbar sein, d.h. sie benötigt keine eigenständige View. Der Code dazu:

```
def destroy
  Book.find(params[:id]).destroy
  redirect_to action: "index"
end
```

Zuerst finden wir mit dem Objekt `param` und dem Parameter `id` das zu löschende Buch und löschen es mit `destroy` aus der Datenbank. Mit dem Aufruf von `redirect_to` kehren wir zur View `index` zurück.

Subjects-Controller

Die folgenden Methoden werden im Books-Controller erfasst, Datei `app/controller/subjects_controller.rb`.

index-Methode

Die `index`-Methode soll alle erfassten Fachgebiete ausgeben, sie sieht wie folgt aus:

```
def index
  @subjects = Subject.all
end
```

Die URL zur View soll wie folgt aussehen: <http://localhost:3000/subjects>

show-Methode

Die `show`-Methode der Fachgebiete soll nicht die Details eines Fachgebietes anzeigen, da wir nur den Namen erfassen. Anstatt dessen sollen alle Bücher angezeigt werden, die diesem Fachgebiet zugeordnet sind. Die Methode sieht wie folgt aus:

```
def show
  @subject = Subject.find(params[:id])
end
```

Mit `Subject.find(params[:id])` wird ein Fachgebiets-Objekt zurückgegeben, das neben dem Fachgebiet auch alle Bücher-Objekte enthält, die diesem Fachgebiet zugeordnet sind. Mit `@subject.books.each` kann die Liste der Bücher durchlaufen werden.

Die URL zur View soll wie folgt aussehen: <http://localhost:3000/subjects/show/:id>Hier Text eingeben



Praktische Arbeit

Aufgabe 1

Erstellen Sie alle oben vorgestellten Methoden in den beiden Controllern. Wenn Sie den Rails-Server beenden und neu starten, werden allfällige Programmierfehler angezeigt. Korrigieren Sie diese, bis der Server fehlerfrei läuft.

Aufgabe 2

Beim Erstellen der Tabelle `subjects` haben wir direkt Daten erfasst (Kapitel *Eine Rails Migration erstellen*, Seite 5):

```
1 class CreateSubjects < ActiveRecord::Migration[5.2]
2   def change
3     create_table :subjects do |t|
4       t.string :name
5       t.timestamps
6     end
7
8     Subject.create name: "Rails"
9     Subject.create name: "Linux"
10    Subject.create name: "SQL"
11    Subject.create name: "Python"
12    Subject.create name: "Bash"
13  end
14 end
```

Die Fachgebiete sind also bereits vorhanden. Erfassen Sie nun folgende Bücher (Book-Objekte) in der Rails-Konsole (`rails c`):

`#<Book id: 1, title: "Rails Guides", price: 21.0, subject_id: 1, description: "">`

`#<Book id: 2, title: "Linux Administration", price: 56.0, subject_id: 2, description: "">`

`#<Book id: 3, title: "SQL Database", price: 12.0, subject_id: 3, description: "">`

`#<Book id: 4, title: "Python for Dummies", price: 9.0, subject_id: 4, description: "">`

`#<Book id: 5, title: "Bourne Again Shell", price: 10.0, subject_id: 5, description: "Bash was first!" >`

`#<Book id: 6, title: "Linux Server", price: 13.0, subject_id: 2, description: "Linux is faster!" >`



Erfassen Sie nur die Attribute `title`, `price`, `subject_id` und `description`, nicht aber die `id`. Diese wird von Rails erstellt und verwaltet.

Aufgabe 3

Statt Daten in der Rails-Konsole zu erfassen, können Sie diese mit der Rails-Migration direkt beim Erstellen der Tabelle erfassen.

1. Welches Kommando ist notwendig, um eine Migration für das Model *Book* zu erstellen?
2. In welcher Datei und Methode können Sie die Datenerfassung der Bücher programmieren?
3. Wie lautet der entsprechende Programmcode? (Führen Sie mind. 3 Bücher auf.)
4. Mit welchem Kommando wird der Programmcode ausgeführt? Dabei werden alle offenen Änderungen seit der letzten Migration durchgeführt.

Schreiben Sie Ihre Lösung auf dieses Blatt, da Sie die 6 Bücher ja bereits in Aufgabe 2 erfasst haben.

3.1

Rails generate migration books

3.2

myapp/db/migrate/<number>_books.rb in der Methode change

3.3

3.4 rails:db:migrate