



**MODUL**

**151**

TEIL

1

:

ACTIVE RECORD

Ralph Maurer

# Inhaltsverzeichnis AB151-01

## Modul 133: Datenbanken in Webapplikationen einbinden

Teil 1: Active Record

Inhaltsverzeichnis AB151-01 ......................................................................................................... 1

Modul 133: Datenbanken in Webapplikationen einbinden .................................................... 1

Active Record ................................................................................................................................. 2

ORM - Object-realational mapping ............................................................................................... 2

Konventionen über Konfiguration ............................................................................................... 3

Namenskonvention ................................................................................................................... 3

Schema Konventionen .............................................................................................................. 6

Active Record Modell erzeugen .................................................................................................... 7

CRUD: Daten lesen und schreiben ................................................................................................ 8

Objekt erstellen (Create) ........................................................................................................... 8

Lesen (Read) .............................................................................................................................. 8

Aktualisieren (Update) .............................................................................................................. 9

Löschen (Delete) ........................................................................................................................ 9

Validierungen .............................................................................................................................. 10

Wieso sind Validierungen wichtig? ......................................................................................... 10 Wann werden Validierungen automatisch gemacht? ........................................................... 11 valid? oder invalid? ................................................................................................................. 11 acceptance .............................................................................................................................. 13 validates\_associated .............................................................................................................. 13 confirmation ............................................................................................................................ 13 length ....................................................................................................................................... 14 numericality ............................................................................................................................ 14 presence .................................................................................................................................. 15

Modell Assoziationen .................................................................................................................. 16

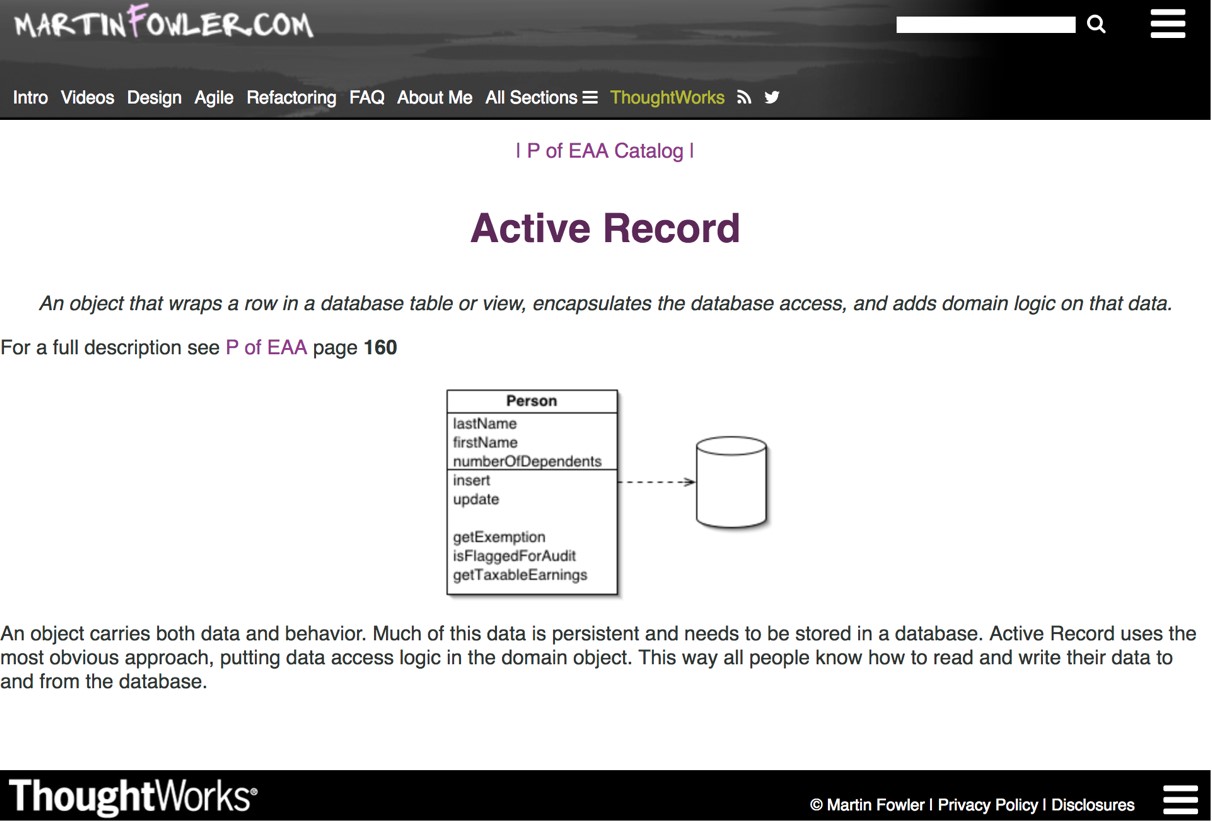
1:1 Beziehungen ...................................................................................................................... 16 1:n Beziehungen ...................................................................................................................... 17 n:n Beziehungen ...................................................................................................................... 18 Auftrag: Quicknote AB151-01 ...................................................................................................... 19

Auftrag: Entwicklungsumgebung für INSTAGRAM als Rails-App: ............................................. 20

# Active Record

In Rails ist der Active Record das M in MVC. Das Model speichert alle Daten und erleichtert die Erstellung und Verwendung von Business objects, die dauerhaft gespeichert werden müssen. Der Active Record ist ein Entwurfsmuster (Software Pattern), das allgemeingültig und abschliessend das Handling eines Datenobjekts beschreibt. Grundsätzlich betrachten wir eine

Zeile in einer Tabelle als Objekt. Martin Fowler, der Autor von dem Software Pattern «Active Record» - beschreibt seine Arbeit unter https://www.martinfowler.com/eaaCatalog/activeRecord.html wie folgt:



Grundsätzlich gilt, dass für ein Objekt, das eine Zeile in einer Datenbanktabelle ist, der Zugriff und die Methoden (Logik) gekapselt und vereinheitlicht sind. Ein Objekt besteht aus Daten und den Methoden, die auch das Verhalten des Objekts ausmachen. Martin Fowler hat gezeigt, dass viele Daten persistent (unveränderbar) sind und auch dauerhaft gespeichert werden müssen. Active Record ist nun ein Mechanismus für den Datenzugriff, die Datenbearbeitung und Datenspeicherung von Objekten.

# ORM - Object-realational mapping

Object-relational mapping- kurz ORM - ist eine Technik, die Objekte einer Anwendung mit Tabellen in einem relationalen Datenbank-Management-System verbindet. Mit ORM können die Eigenschaften und Beziehungen der Objekte in einer Anwendung einfach gespeichert und aus einer Datenbank abgerufen werden, ohne dass SQL-Anweisungen direkt und mit weniger allgemeinem Datenbankzugriffscode geschrieben werden müssen.

**Active Record stellt mehrere Mechanismen zur Verfügung:**

› Darstellung von Modellen und Daten

› Beziehungen zwischen den Modellen

› Vererbungshierarchie durch verwandte Modelle

› Datenvalidierung

› Datenbankoperation auf objekt-orientierte Art und Weise

# Konventionen über Konfiguration

Wenn Sie Anwendungen mit anderen Programmiersprachen oder Frameworks schreiben, müssen Sie möglicherweise viel Konfigurationscode erstellen. Dies gilt insbesondere für ORMFrameworkks. Wenn Sie jedoch die von Rails übernommenen Konventionen befolgen, müssen Sie beim Erstellen von Active Record-Modellen sehr wenig Konfiguration (in einigen Fällen überhaupt keine Konfiguration) schreiben. Die Idee ist, dass wenn Konventionen strikte eingehalten werden, sollte dieser Standard, explizite Konfigurationen erübrigen. Explizite Konfiguration sind nur in den Fällen erforderlich, in denen Sie der Standardkonvention nicht folgen können.

## Namenskonvention

Active Record verwendet standardmässig einige Namenskonventionen, um herauszufinden, wie die Zuordnung zwischen Modellen und Datenbanktabellen erstellt werden soll. In Rails werden Tabellen pluralisierst, um die entsprechende Datenbanktabellen zu finden. Für eine Klasse Book sollten Sie eine Datenbanktabelle namens books anwenden:

**Klasse**

workspace/myapp/book.rb

Innerhalb der Datei book.rb wird die Klasse Book mit grossem Erstbuchstaben definiert.

|  |
| --- |
| class Book < ApplicationRecord  end |

**Datenbanktabelle mit Hilfe der** rails console

2.4.2 :001 > Book.connection

…

2.4.2 :002 > Book

=> Book(id: integer, title: string, price: float, subject\_id: integer, description: text, created\_at: datetime)

2.4.2 :003 > Book.all

Book Load (0.3ms) SELECT "books".\* FROM "books" LIMIT ? [["LIMIT", 11]]

=> #<ActiveRecord::Relation [#<Book id: 2, title: "Rails Guides", price: 213.0, subject\_id: 1, description: "Ruby on Rails!!!", created\_at: "2018-05-02 11:57:22">,

#<Book id: 3, title: "Linux Administration", price: 56.0, subject\_id: 2, description: "", created\_at: "2018-05-02 11:57:42">,

#<Book id: 4, title: "SQL Database", price: 12.0, subject\_id: 3, description: "asdasd", created\_at: "2018-05-02 11:57:55">,

…

**Aufgabe 1**

Beantworten Sie folgende Fragen unter Berücksichtigung der Namenskonvention:

1. Was bedeutet 2.4.2 :001 > Book.connection ?
2. Was bedeutet 2.4.2 :001 > Book ?
3. Was bedeutet 2.4.2 :001 > Book.all ?
   * + 1. Verbindung zu Model «Book»
       2. Model von Book wird angezeigt (Attribute mit Datentypen)
       3. Alle Datensätze werden ausgegeben

Die Pluralisierungsmechanismen von Rails sind sehr leistungsfähig und in der Lage, sowohl reguläre als auch irreguläre Wörter zu pluralisieren (und zu vereinzeln). Wenn Sie

Klassennamen verwenden, die aus zwei oder mehr Wörtern bestehen, sollte der

Modellklassenname gemäss Ruby-Konvention in CamelCase geschrieben werden, während der Tabellenname (auch Schema genannt) durch Unterstriche getrennten Wörter definiert wird.

**Beispiel:**

› Klasse: BookClub

› Tabelle: book\_clubs

Bei der Klasse werden die Erstbuchstaben aller Wörter grossgeschrieben.

**Aufgabe 2**

Vervollständigen Sie folgende Tabelle. Beachten Sie das alle Modell- und Klassennamen Englisch geschrieben sind.

|  |  |  |
| --- | --- | --- |
| **Aufgabe** | **Modell / Klasse** | **Tabellenname / Schema** |
|  | Budget | budget |
|  | BudgetPosition | budget\_positions |
|  | Person | people |
|  | Mouse | mice |
|  | FootballTeam | football\_teams |
|  | City | cities |
|  | Berry | berries |
|  | ClockOfChurch | clock\_of\_chruches |
|  | Fox | foxes |
|  | Knife | knives |
|  | Spoof | spoofs |
|  | Scarf | scarves |
|  | Neurose | neuroses |
|  | Analysis | analyses |
|  | Crise | crises |
|  | Zero | zeros |

## Schema Konventionen

Abhängig vom Zweck eines Attributs verwendet Active Record unterschiedliche Namenskonventionen für Attribute in Datenbanktabellen. **Fremdschlüssel**

singularized\_table\_name\_id (z.B. book\_id, subject\_id)

Active Record erkennt diese Schreibweise als Beziehung zwischen Tabellen.

**Primärschlüssel**

Active Record verwendet standardmäßig eine ganzzahlige Spalte mit dem Namen id als Primärschlüssel der Tabelle.

Wenn Active Record Migration zum Erstellen Ihrer Tabellen verwendet wird, wird diese Spalte automatisch erstellt. Genauso die Attribute:

|  |  |
| --- | --- |
| created\_at | Ermittelt automatisch das aktuelle Datum und die Uhrzeit, wann der Datensatz erstellt wurde. |
| updated\_at | Bei jeder Aktualisierung des Datensatzes wird automatisch das aktuelle Datum und die aktuelle Uhrzeit eingestellt. |
| lock\_version | Fügt einem Modell eine optimistische Sperre hinzu. |
| type | Gibt an, dass das Modell die Vererbung einzelner Tabellen verwendet. |
| (association\_name)\_type | Speichert den Typen für polymorphe Assoziationen. |
| (table\_name)\_count | Wird verwendet, um die Anzahl Objekte in einer Beziehung zu zählen.  **Beispiel:** Eine Spalte "comments\_count" in einer Article-  Klasse, die viele Instanzen von "Comment" enthält, speichert die Anzahl der vorhandenen Kommentare für jeden Artikel. |

# Active Record Modell erzeugen

Es ist sehr einfach ein Active Record Modell zu erzeugen. Man muss eine Subklasse von der Klasse ActiveRecord erstellen:

workspace/myapp/book.rb

|  |
| --- |
| class Book < ApplicationRecord  end |

Angenommen wir haben eine einfache Migration erstellt, welche eine Tabelle erstellt (Beispiel aus Modul 133).

workspace/myapp/db/migratNNNNNNNNNN\_books.rb

|  |
| --- |
| class Books < ActiveRecord::Migration[5.1] def self.up  create\_table :books do |t|  t.column :title, :string, :limit => 32, :null => false  t.column :price, :float  t.column :subject\_id, :integer  t.column :description, :text end end  def self.down drop\_table :books end end |

Die Migration erstellt eine Tabelle books mit den Attributen title, price, subject\_id, description.

Ein Objekt Book wird nun wie folgt erstellt:

b = Book.new

b.title = 'Ein Buch'

b.price = 12.10

b.subject\_id = 9

b.descripton = 'Eine Buchbeschreibung'

b.save

# CRUD: Daten lesen und schreiben

## Objekt erstellen (Create)

**Aufgabe 3**

Erstellen Sie ein Objekt Book mit der folgendem Active Record entspricht:

#<Book title: "Linux Server", price: 13.0, subject\_id: 2, description: "Linux is faster"> Nutzen Sie die create-Methode:

|  |
| --- |
| Book.create(title : «Linux Server ») |
|  |
|  |
|  |
|  |
|  |

Nutzen Sie für den Active Record

#<Book title: "Ruby Book", price: 21.00, subject\_id: 8, description: "Simple as that"> die new-Methode sowie save-Methode:

|  |
| --- |
| B = Book.new  B.title = «Ruby Book »  b.price = 21 .00 |
|  |
|  |
|  |
|  |
|  |

Lesen (Read)

Active Record bietet eine umfangreiche API für den Zugriff auf Daten in einer Datenbank. Im Folgenden finden Sie einige Beispiele für verschiedene Datenzugriffsmethoden, die von Active Record bereitgestellt werden.

|  |
| --- |
| # alle Bücher ausgeben b = Book.all  # Erstes Buch in Modell ausgeben b = Book.first  # Nach dem Buch Rails suchen  railsbook = Book.find\_by(title: 'Rails')  # oder  railsbook = Book.find\_by\_title('Rails')  # Alle Bücher billiger als 100 sortiert nach Preis  Book.where("price < 102").order(:price) |

Eine nahezu vollständige Beschreibung des Querying finden Sie unter sftp.iet-gibb.ch/sh-modules/iet\_151/03\_Arbeitsblaetter/ AB151-

01\_ActiveRecordQuery\_Interface.pdf

Quelle: https://guides.rubyonrails.org/active\_record\_querying.html

## Aktualisieren (Update)

Sobald ein Active Record ausgewählt ist, können seine Werte einfach verändert und im Modell gespeichert werden:

b = Book.find\_by(title: 'Linux Server')

b.title = 'Advanced Linux Server'

b.price = 25.0

b.save

Man kann auch kürzer vorgehen und neue Werte mit der update-Methode aktualisieren.

b = Book.find\_by(title: 'Linux Server')

b.update(title: 'Advanced Linux Server', price: 25.0)

b.save

## Löschen (Delete)

Wenn ein Active Record ein Objekt enthält, kann man dieses löschen:

b = Book.find\_by(name: 'Rails')

b.destroy

Es können mehrere Objekte gelöscht werden:

Book.where("price < 102").destroy\_all

Alle Objekte löschen:

Book.destroy\_all

# Validierungen

Mit Active Record können Sie den Status eines Objekts überprüfen, bevor es in die Datenbank geschrieben wird. Es gibt mehrere Methoden, mit denen Sie Ihre Objekte überprüfen können. Geprüft werden kann u.a. ob ein Attributwert nicht leer ist, eindeutig oder erst gar nicht in der Datenbank vorhanden ist, ein bestimmtes Format eingehalten wird und vieles mehr.

Die Validierung wird immer bei der Speicherung in eine Datenbank überprüft. Die Validierungsmethoden geben false zurück, wenn die Validierung fehlschlägt. Zur Veranschaulichung:

class Book < ApplicationRecord validates :title, presence: true end

b = Book.new

b.save # => false

b.save! # => ActiveRecord::RecordInvalid: Validation failed: Name can't be blank

oder

Book.create(title: "Book of Ruby").valid? # => true Book.create(title: nil).valid? # => false

Wieso sind Validierungen wichtig?

Mit Validierungen wird sichergestellt, dass nur gültige Daten in der Datenbank gespeichert werden. Beispielsweise kann es für eine Anwendung wichtig sein, sicherzustellen, dass jeder Benutzer eine gültige E-Mail-Adresse und Postanschrift angibt.

Validierungen sind datenbankunabhängig, im Sinne, dass Sie auf verschieden Datenbanken anwendbar sind. Validierungen können nicht von Endbenutzern umgangen werden und sind für Entwicklerinnen und Entwickler bequem zu testen und zu warten.

Es gibt verschiedene Möglichkeiten, Daten zu validieren, bevor sie in einer Datenbank gespeichert werden:

1. nativer Datenbankeinschränkungen
2. clientseitiger Validierungen
3. Validierungen auf Controller-Ebene.

Hier ist eine Zusammenfassung der Vor- und Nachteile:

› Datenbankeinschränkungen und / oder gespeicherte Prozeduren machen die Validierungsmechanismen datenbankabhängig und können das Testen und die Wartung erschweren. Wenn Ihre Datenbank jedoch von anderen Anwendungen verwendet wird, empfiehlt es sich, einige Einschränkungen auf Datenbankebene zu verwenden. Darüber hinaus können Validierungen auf Datenbankebene einige Dinge sicher handhaben (z. B. die Eindeutigkeit in stark genutzten Tabellen), die andernfalls nur schwer zu implementieren sind.

› Clientseitige Validierungen können nützlich sein, sind aber im Allgemeinen unzuverlässig. Wenn Validierungen mit JavaScript implementiert werden, können diese umgangen werden, z.B. wenn JavaScript im Browser des Benutzers deaktiviert ist. In Kombination mit anderen Techniken kann die clientseitige Validierung jedoch eine bequeme Methode sein, um Benutzern unmittelbares Feedback zu geben.

› Validierungen auf Controller-Ebene können verlockend sein, werden aber oft unübersichtlich und sind schwer zu testen und zu warten.

Nach Ansicht des Rails-Teams sind Validierungen auf Modellebene in den meisten Fällen am besten geeignet.

Wann werden Validierungen automatisch gemacht?

Die folgenden Methoden triggern automatisch eine Validierung bevor das Objekt im Modell gespeichert wird. Gespeichert werden nur valide Objekte.

› create › create! › save › save!

› update › update!

Der Unterschied zwischen create und create! resp. save und save! oder update und update! hat mit Validierungen zu tun. Mit einem Ausrufezeichen wird eine Bang-Version der Methode ausgeführt. Eine Bangversion retourniert eine Fehlermeldung, während nonBangversionen nur true oder false zurückgeben.

Folgende Methoden machen keine Validierung und speichern Objekte im Modell ohne Prüfung ab. Deshalb sollten diese Methoden mit Vorsicht angewandt werden:

› decrement! › decrement\_counter › increment! › increment\_counter › toggle!

› touch

› update\_all

› update\_attribute

› update\_column

› update\_columns

› update\_counters

Man kann auch bei validierenden Methoden die Prüfung explizit deaktivieren:

› save(validate: false)

valid? oder invalid?

Bevor ein Active Record gespeichert wird, können Sie mit den Prädikaten valid? oder invalid? die Validierung eines Objekts prüfen.

|  |
| --- |
| class Book < ApplicationRecord validates :title, presence: true end    Book.create(name: "John Doe").valid? # => true  Person.create(name: nil).valid? # => false |

invalid? ist die Umkehrung von valid?. invalid? retourniert true, falls Fehler gefunden wurden und false falls nicht.

Nachdem Active Record die Validierungen durchgeführt hat, kann auf alle gefundenen Fehler über die Instanzmethode errors.messages zugegriffen werden. Per Definition ist ein Objekt gültig, wenn diese Sammlung nach dem Ausführen von Validierungen leer ist.

Beachten Sie, dass ein Objekt, das mit new instanziiert wurde, auch dann keine Fehler meldet, wenn es technisch ungültig ist, da Validierungen automatisch nur beim Speichern des Objekts ausgeführt werden, z. B. mit den Methoden create oder save.

|  |
| --- |
| class Book < ApplicationRecord validates :title, presence: true end    >> b = Book.new  # => #<Book id: nil, title: nil>  >> b.errors.messages  # => {}    >> b.valid?  # => false  >> b.errors.messages  # => {name:["can't be blank"]}    >> b = Book.create  # => #<Book id: nil, title: nil>  >> b.errors.messages  # => {name:["can't be blank"]}    >> b.save  # => false    >> b.save!  # => ActiveRecord::RecordInvalid: Validation failed: title can't be blank    >> Book.create!  # => ActiveRecord::RecordInvalid: Validation failed: title can't be blank |

## acceptance

Diese Methode kontrolliert, ob eine Checkbox in einem Formular aktiv ist. Dies wird u.a. bei «I agree to terms of service» angewandt:

class Person < ApplicationRecord validates :terms\_of\_service, acceptance: true end

Der oben aufgeführte Code gibt true zurück, falls die Checkbox «terms\_of\_service» nicht nil ist. Man kann aber acceptance bei der Validierung auch individuell konfigurieren:

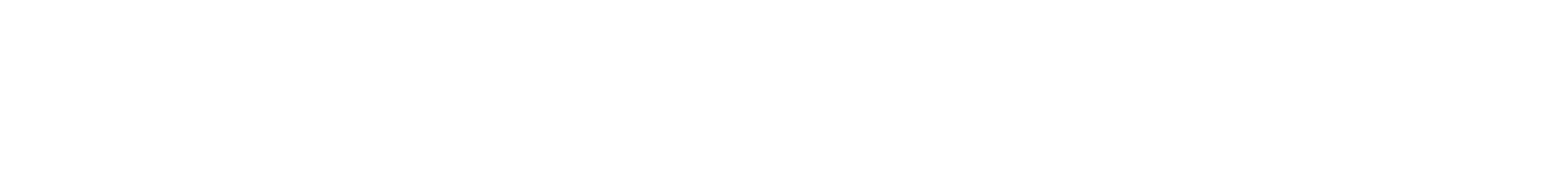
class Person < ApplicationRecord

validates :terms\_of\_service, acceptance: { message: have to be abided' } end

## validates\_associated

Sie sollten diese Validierung verwenden, wenn Ihr Modell Assoziationen mit anderen Modellen aufweist und diese ebenfalls validiert werden müssen. Wenn Sie versuchen, ein Objekt zu speichern, wird valid? auf jedes der zugehörigen Objekte aufgerufen.

class Subject < ApplicationRecord has\_many :books validates\_associated :books end



Verwenden Sie

validates\_associated

nicht an beiden Enden Ihrer

Assoziation

.

Die

Validierungen

würden sich gegenseitig in einer Endlosschleife

aufrufen

.

## confirmation

Sie sollten diese Validierung verwenden, wenn Sie zwei Textfelder haben, die genau den gleichen Inhalt erhalten sollen. Zum Beispiel möchten Sie eine E-Mail-Adresse oder ein Passwort bestätigen. Diese Validierung erstellt ein virtuelles Attribut, dessen Name der Name des Felds ist, das mit confirmation bestätigt werden muss.

class Person < ApplicationRecord validates :email, confirmation: true end

In der View können Sie confirmation wie folgt einsetzen:

<%= text\_field :person, :email %>

<%= text\_field :person, :email\_confirmation %>

## length

Diese Validierung kontrolliert die Länge der Attributwerte. Es bietet eine Vielzahl von Optionen, sodass Sie Längenbeschränkungen auf verschiedene Arten festlegen können:

class Book < ApplicationRecord validates :name, length: { minimum: 2 } validates :description, length: { maximum: 500 } validates :password\_for\_access, length: { in: 6..20 } validates :ISBN, length: { is: 10 } end

Die möglichen Längenbeschränkungen sind:

› :minimum - Das Attribut darf nicht kleiner als die angegebene Länge sein.

› :maximum - Das Attribut darf nicht länger als die angegebene Länge sein.

› :in (oder: innerhalb) - Die Attributlänge muss in einem bestimmten Intervall enthalten sein. Der Wert für diese Option muss ein Bereich sein.

› :is - Die Attributlänge muss dem angegebenen Wert entsprechen.

## numericality

Diese Validierung prüft, ob ein Wert rein nummerisch ist. Dabei werden integer und float geprüft. Will man nur einen integer zulassen, empfiehlt sich die Verwendung von :only\_integer.

|  |
| --- |
| class Book < ApplicationRecord validates :price, numericality: true  validates :pages, numericality: { only\_integer: true } end |

Neben :only\_integer akzeptiert diese Validierung auch die folgenden Optionen:

› :greater\_than - Gibt an, dass der Wert grösser als der angegebene Wert sein muss. Die Standardfehlermeldung für diese Option lautet "muss grösser als %{count}" sein.

› :greater\_than\_or\_equal\_to - Gibt an, dass der Wert grösser oder gleich dem angegebenen Wert sein muss. Die Standardfehlermeldung für diese Option lautet "muss grösser oder gleich %{count}" sein.

› :equal\_to - Gibt an, dass der Wert dem angegebenen Wert entsprechen muss. Die Standardfehlermeldung für diese Option lautet "muss gleich %{count}" sein.

› :less\_than - Gibt an, dass der Wert kleiner als der angegebene Wert sein muss. Die Standardfehlermeldung für diese Option lautet "Muss kleiner als %{count}" sein.

› :less\_than\_or\_equal\_to - Gibt an, dass der Wert kleiner oder gleich dem angegebenen Wert sein muss. Die Standardfehlermeldung für diese Option lautet "muss kleiner oder gleich %{count}" sein.

› :other\_than - Gibt an, dass der Wert anders als der angegebene Wert sein muss. Die Standardfehlermeldung für diese Option lautet "muss anders als %{count}" sein.

› :ungerade - Gibt an, dass der Wert eine ungerade Zahl sein muss, wenn er auf "true" gesetzt ist. Die Standardfehlermeldung für diese Option lautet "muss ungerade sein".

› :even - Gibt an, dass der Wert eine gerade Zahl sein muss, wenn er auf "True" festgelegt ist. Die Standardfehlermeldung für diese Option lautet "muss gerade sein".

|  |  |
| --- | --- |
| AB151-01 |  |
| Seite 15/20 | Diese Validierung prüft, ob ein angegebener Wert nicht leer ist, angewandt wird die blank? Methode. |

## presence

class Book < ApplicationRecord

validates :title, :price, :description, presence: true end

Wenn Sie eine Assoziation prüfen wollen, müssen Sie testen ob ein assoziiertes Objekt vorhanden ist und nicht der Fremdschlüssel:

class Book < ApplicationRecord belongs\_to :subject

validates :subject, presence: true end

Weitere Validierungen finden Sie unter: sftp.iet-gibb.ch/sh-

modules/iet\_151/03\_Arbeitsblaetter/ AB151-01\_ActiveRecordValidations.pdf

Quelle: https://guides.rubyonrails.org/active\_record\_validations.html

# Modell Assoziationen

Wir haben gesehen, dass Rails viele Modell-Settings über die Namenskonvention regelt. Die Frage, die im Raum steht ist: Wie behandelt Rails Assoziationen und Vererbung? Überraschenderweise lautet die Antwort wieder über Namenskonventionen. Assoziationen werden dabei immer im Model beschrieben 1:1 Beziehungen

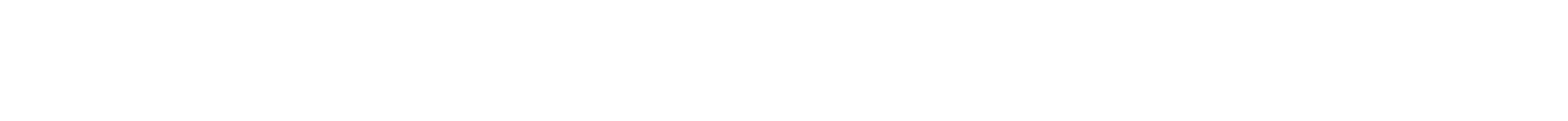
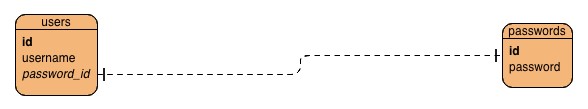
Um eine 1:1 Beziehung zu realisieren müssen die beteiligten Tabellen bzw. Modelle zunächst der Namenskonvention folgen. Ausserdem muss in einer der beteiligten Tabellen ein Fremdschlüssel existieren. Dieser setzt sich per Default aus dem Namen der Tabelle (Singular) und dem Suffix „id“ zusammen. Auf Datenbank-Seite ist weiter nichts zu tun. Auf Modell-Ebene dagegen muss die Assoziation über Klassenmethoden deklariert werden. Dafür stehen bei 1:1 Beziehungen die Methoden belongs\_to und has\_one zur Verfügung. Diesen werden dann als Parameter die Namen der Assoziation übergeben (Singular der Tabellen).

class User < ActiveRecord::Base

belongs

\_

to :password



Der Unterschied zwischen den beiden Methoden besteht darin, dass

belongs\_to

die

Tabelle mit dem Fremdschlüssel anzeigt

.

end

class Password < ActiveRecord::Base

has\_one :user end

Sind die Tabellen und die Modelle entsprechend erzeugt, ist das ORM auch schon komplett konfiguriert. Auf Code-Ebene navigiert man nun wie gewohnt auf den Modellen. Für das obige Beispiel muss dafür zunächst über die zur Verfügung gestellten Such- bzw.

Erzeugungsmethoden eine Instanz an eine Variable gebunden werden. Anschliessend erhält man über die Punktnotation die assoziierten Objekte:

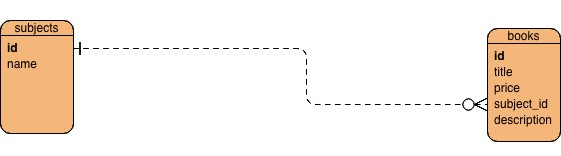
user.password

oder

password.user

## 1:n Beziehungen

Die Behandlung von Assoziationen mit nur einem mehrfachen Ende unterscheidet sich bei Ruby on Rails nur unwesentlich von der des kleinen Bruders, den 1:1 Beziehungen. Natürlich müssen jedoch auch hier die beteiligten Tabellen und Modelle den Konventionen folgen. Die Tabellenstruktur ist dabei analog zur einfachen Assoziation aufgebaut. Der Fremdschlüssel befindet sich selbstverständlich auf der N-Seite und besteht wiederum aus dem Tabellennamen plus das Suffix id.



class Subject < ActiveRecord::Base has\_many :books end

|  |
| --- |
| class Book < ActiveRecord::Base  belongs\_to :subject end |

Auf Modellebene ändert sich lediglich die Deklaration der mehrfachen Assoziation. Anstatt einem has\_one symbolisiert nun ein has\_many auf der 1-Seite der Beziehung, dass dieses Modell viele Assoziationspartner besitzt. Die eindeutige Zuordnung der N-Seite wird dagegen weiterhin durch ein belongs\_to angezeigt. Für die Navigation bzw. Zuweisung sind die Veränderung ebenso einseitig. Auf Modellebene ändert sich lediglich die Deklaration der mehrfachen Assoziation. Anstatt einem has\_one symbolisiert nun ein has\_many auf der 1Seite der Beziehung, dass dieses Modell viele Assoziationspartner besitzt. Die eindeutige Zuordnung der N-Seite wird dagegen weiterhin durch ein belongs\_to angezeigt. Für die Navigation bzw. Zuweisung sind die Veränderung ebenso einseitig.

Bei der Initialisierung der Beziehung wird diesmal als Parameter allerdings ein Array an Domain-Objekten und nicht ein einzelnes erwartet:

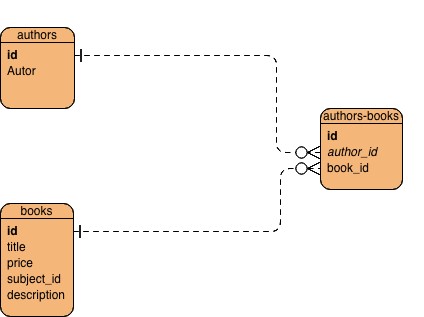
subject.books = [ book1, book2, book3, book4 ]

Die spezielle Kollektion erlaubt in Array-Manier ebenfalls das Hinzufügen eines einzelnen Objektes zur Kollektion:

subject.books << book5

## n:n Beziehungen

Die dritte der möglichen Beziehungstypen sind die n:n Beziehungen oder kompex zu komplex Assoziation. So überrascht es nicht, dass ein wenig mehr Aufwand notwendig ist, um eine solche komplex-komplex Assoziation in Rails zu realisieren. Doch zunächst steht wiederum das Datenbank-Schema im Focus. In relationalen Datenbanken werden n:n Beziehungen über eine dritte Zwischentabelle realisiert. Diese Tabelle beinhaltet jeweils den Primärschlüssel der beteiligten Tabellen als Fremdschlüssel. Per Konvention werden die Schlüssel wie zuvor aus dem Singular des Tabellennamens und dem Suffix „id“ zusammengestzt. Der Name für die Zwischentabelle selber wird aus den beiden beteiligten Tabellennamen gebildet. Die Reihenfolge ist dabei alphabetisch zu wählen.



class Book < ActiveRecord::Base has\_and\_belongs\_to\_many :authors end

class Author < ActiveRecord::Base has\_and\_belongs\_to\_many :books end

Auf Modellebene werden n:n Assoziationen über die Klassenmethode

has\_and\_belongs\_to\_many deklariert. Die Navigation bzw. Zuweisung gestaltet sich nun genauso, wie für die N-Seite bei 1:n Beziehungen, mit dem Unterschied, dass dies für beide

Assoziationsenden gilt. Für beide bekommt man also eine Kollektion geliefert, die u.a. das Löschen, Erzeugen und Iterieren von Assoziationsobjeken ermöglicht

# Auftrag: Quicknote AB151-01

Alle Aufträge des Typen Quicknote sind Bestandteil der Bewertung. Erstelle Sie eine Quicknote von max. 4 A4-Seiten und geben Sie diese in PDF Form gemäss Zeitangabe der Lehrperson ab. Angedacht sind max. 4 Lektionen seit Abgabe dieses Dokumentes.

**Titel der Quicknote:** *Klasse\_Name\_Vorname\_QN\_AB151-01.pdf* **Beispiel:** INF17R\_Maurer\_Ralph\_QN\_AB151-01.pdf

Was beinhaltet die Quicknote AB151-01?

Zusammenfassung AB151-1

Die Quicknote soll eine kurze prägnante Zusammenfassung des Dokuments AB151-01 beinhalten und einen Überblick über die vorgestellten Techniken, Methoden und Konzepte beinhalten.

Achten Sie darauf, dass alle wesentliche Themen vollständig erwähnt und wenn möglich in Zusammenhang gebracht sind.

Beantworten Sie folgende Fragen zu Anwendungszweck und Selbstreflexion:

Anwendungszweck:

1. Wie und wo können die vorgestellten Techniken, Methoden und Konzepte in einer Rails-App angewandt werden?
2. Was sind Vorteile und was sind Nachteile?

Selbstreflexion:

1. Was habe ich gelernt?
2. Was hat mich behindert?
3. Was habe ich nicht verstanden?
4. Was kann ich beim Studium besser machen?

Lösungen der Aufgaben:

Die Quicknote muss die Lösungen der Aufgabe 1 (S. 4), Aufgabe 2 (S.5) und Aufgabe 3 (S. 8) sowie ein Kurzbeschrieb Ihrer Lösung für die Inbetriebnahme der Entwicklungsumgebung für Instagram als Rails-App (letzte Seite) beinhalten.

Abschliessende Reflexion über das Gelernte:

Schreiben Sie eine persönliche Schlussfolgerung über den Lerninhalt in 2-3 Sätzen.

Auftrag: Entwicklungsumgebung für INSTAGRAM als Rails-App vorbereiten:

Wir werden in Modul 151 Instagram als Railsanwendung nachbauen.

Erstellen Sie auf bmLP1 eine neue Rails-App mit Namen instagram. Achten Sie darauf dass der Localhost auf Port 4000 läuft. Auf Port 3000 soll weiterhin myapp aus Modul 131 laufen.

