Ruby On Rails course

Active Record

The persistance layer

The **M** in MVC

What is an ORM?

Object relational mapping is a programming technique for converting data between object-oriented programming languages and SQL databases.

ORM in practice

```
User.find_by(id: 1)
```

```
SELECT * FROM users WHERE users.id = 1 LIMIT 1
```

```
#<User id: 1, email: "damir@gmail.com", name: 'Damir'>
```

ActiveRecord

Rails ORM system

ActiveRecord

- An Object Relational Mapper built in Ruby
- Comes by default in Rails

ActiveRecord is SQL database agnostic

Database support for:

- Postgres
- MySQL
- Sqlite
- ...
- ~~MongoDB~~

Let’s create a database

We'll be using sqlite3

Self contained database

Configuration: config／database.yml

```
default: &default
  adapter: sqlite3
  pool: 5
  timeout: 5000

development:
  <<: *default
  database: db/development.sqlite3

test:
  <<: *default
  database: db/test.sqlite3

production:
  <<: *default
  database: db/production.sqlite3
```

Rake task for creating a database

```
rake db:create
```

Create a model

Create a User model

```
rails generate model User email:string
name:string age:integer
```

Two main files we're interested in:

- `app/models/user.rb`
- `db/migrate/20150705121118_create_users.rb`

app/models/user.rb

```ruby
class User < ActiveRecord::Base
end
```

db/migrations/20150705121118_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :email
      t.string :name
      t.integer :age

      t.timestamps null: false
    end
  end
end
```

Naming conventions

Model / Class	Table
User	users
Article	articles
BookingInfo	booking_infos
Person	people

Why would anyone use migrations?

Easier database schema synchronization

Let’s migrate our database

Rake task for migrating our database

```
rake db:migrate
```

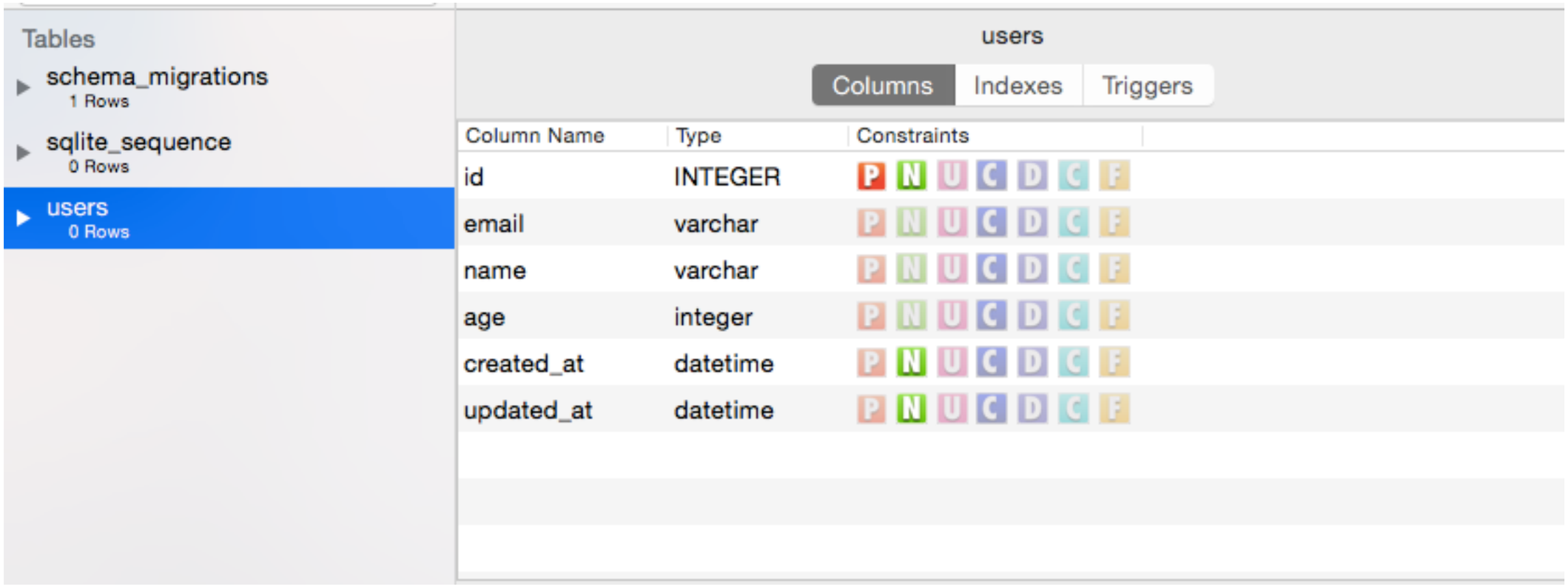
db/schema.rb

```
ActiveRecord::Schema.define(version: 20150705121118) do

  create_table "users" do |t|
    t.string   "email"
    t.string   "name"
    t.integer  "age"
    t.datetime "created_at", null: false
    t.datetime "updated_at", null: false
  end

end
```

View it through a DB Browser..



Default columns

- id
- created_at
- updated_at

Changing table structures

Change column name

```
class ChangeEmailColumnName < ActiveRecord::Migration
  def change
    rename_column :users, :email, :email_address
  end
end
```

Drop table

```
class DropUsers < ActiveRecord::Migration
  def change
    drop_table :users
  end
end
```

CRUD With ActiveRecord

Create–Read–Update–Delete

CREATE
READ
UPDATE
DELETE

Create a new record

```
User.create(email: ‘robert@gmail.com’, name: ‘Robert’)
```

```
INSERT INTO "users" ("email", "name", "created_at",
"updated_at") VALUES ("robert@gmail.com", "Robert",
    "2015-07-05 10:40:04.206071", "2015-07-05
                10:40:04.206071")
```

Create a new record

```
user =
  User.new(email: ‘robert@gmail.com’,
  name: ‘Robert’)

user.save
```

Let’s dive into the rails console..

Rails console

- just like IRB but loads the Rails console
- `rails console` (or simpler `rails c`)

CREATE

READ

UPDATE

DELETE

Find all records

```
User.all
```

```
SELECT * FROM users
```

Find multiple record

```
User.where(age: 25)
```

```
SELECT * FROM users WHERE users.age = 25
```

```
#<User id: 1, email: "damir@gmail.com", age: 25>
#<User id: 3, email: "ivana@gmail.com", age: 25>
#<User id: 9, email: “marko@gmail.com”, age: 25>
```

Find the first record

```
User.first
```

```
SELECT  "users".* FROM "users"  ORDER BY "users"."id"
ASC LIMIT 1
```

Find one record

```
User.find_by(email: 'damir@gmail.com')
```

```
SELECT * FROM users WHERE users.email =
"damir@gmail.com" LIMIT 1
```

```
#<User id: 1, email: "damir@gmail.com", name: 'Damir'>
```

Count the records

```
User.count
```

```
SELECT COUNT(*) FROM users
```

CREATE
READ
UPDATE
DELETE

Update

```
user = User.find_by(email: ‘damir@gmail.com’)

#<User id: 1, email: "damir@gmail.com", age: 26>

user.update(age: 26)
```

```
UPDATE "users" SET "age" = 26, "updated_at" =
"2015-07-05 11:48:47.130524" WHERE "users"."id" = 1
```

Update

```
user = User.find_by(email: ‘damir@gmail.com’)

user.age = 26

user.save
```

CREATE
READ
UPDATE
DELETE

Destroy

```
user = User.find_by(email: ‘damir@gmail.com’)
```

```
#<User id: 1, email: "damir@gmail.com", age: 26>
```

```
user.destroy
```

```
DELETE FROM "users" WHERE "users"."id" = 1
```

Validations

We don’t want users with blank emails or names

```
user = User.create

#<User id: 1, email: nil, name: nil, age: nil>

user.valid? #=> true
```

Presence validations

Let’s add some validations

```
class User < ActiveRecord::Base

  validates :name,  presence: true
  validates :email, presence: true

end
```

We don’t want users with blank emails or names

```
user = User.create

#<User id: nil, email: nil, name: nil, age: nil>

user.valid? #=> false
```

What were the errors?

```
user.errors.any?
# true

user.errors
# :name=>["can't be blank"], :email=>["can't be
blank”]

user.errors.full_messages
# ["Name can't be blank", "Email can't be blank"]
```

Let’s create some valid users

```
user = User.create(name: 'Damir', email:
'damir@gmail.com')
```

```
#<User id: 1, email: ’damir@gmail.com’, name:
’Damir’, age: nil>

user.valid? #=> true
```

Format validations

But users can be created with invalid emails

```
user = User.create(name: 'Damir', email: ‘damir‘)

#<User id: 1, email: ’damir’, name: ’Damir’, age:
nil>

user.valid? #=> true
```

Let’s add email validations

```
class User < ActiveRecord::Base
  VALID_EMAIL_REGEX = /(.+)@(.+){2,}\.(.+){2,}/

  validates :name,  presence: true
  validates :email, presence: true,
                    format: VALID_EMAIL_REGEX
end
```


* not to be used in production

Uniqueness validations

But two users can be created with the same email

```
user = User.create(name: 'Marko', email:
‘ksetovac@gmail.com’)

user.valid? #=> true
```

```
user = User.create(name: 'Ivan', email:
‘ksetovac@gmail.com’)

user.valid? #=> true
```

Let’s add some uniqueness email validations

```
class User < ActiveRecord::Base
  VALID_EMAIL_REGEX = /(.+)@(.+){2,}\.(.+){2,}/

  validates :name,  presence: true
  validates :email, presence: true,
                    format: VALID_EMAIL_REGEX,
                    uniqueness: true
end
```


* not to be used in production

Length validations

We don’t want users with names shorter than 2 characters and longer than 25

```
class User < ActiveRecord::Base
  VALID_EMAIL_REGEX = /(.+)@(.+){2,}\.(.+){2,}/

  validates :name,  presence: true,
                    length: { minimum: 2,
                              maximum: 25 }

  validates :email, presence: true,
                    format: VALID_EMAIL_REGEX,
                    uniqueness: true
end
```

Numerical validations

The age should be numerical

```
class User < ActiveRecord::Base
  VALID_EMAIL_REGEX = /(.+)@(.+){2,}\.(.+){2,}/

  validates :name,  presence: true,
                    length: { minimum: 2,
                              maximum: 25 }

  validates :email, presence: true,
                    format: VALID_EMAIL_REGEX,
                    uniqueness: true

  validates :age, numericality: true,
                  allow_blank: true
end
```

And many many more...

http://guides.rubyonrails.org/active_record_validations.html

Associations

Let’s create an additional Recipe model

```
rails generate model Recipe title:string
content:text user_id:integer
```

User & Recipe classes

```ruby
class User < ActiveRecord::Base
end
```

```ruby
class Recipe < ActiveRecord::Base
end
```

has_many

Let’s create some recipes..

```
Recipe.create(title: ‘Tasty Rubies', content: ‘Boil
20 minutes and then …’, user_id: 1)

Recipe.create(title: ‘Spicy Rubies', content: ‘No
compiling need, just put in water..’, user_id: 1)
```

Let’s fetch all the recipes by User with id: 1

```
Recipe.where(user_id: 1)
```

```
[#<Recipe id: 1, title: ‘Tasty Rubies', content: ‘Boil 20 minutes and
then …’, user_id: 1>,
 #<Recipe id: 2, title: ‘Spicy Rubies', content: ‘No compiling need,
just put in water..’, user_id: 1>]
```

But theres a nicer way to do this..

```ruby
class User < ActiveRecord::Base
  has_many :recipes
end
```

But theres a nicer way to do this..

```
user = User.find_by(id: 1)

user.recipes
```

```
SELECT * FROM "recipes" WHERE "recipes"."user_id"
= 1
```

belongs_to

Let’s create the reverse association

```
class Recipe < ActiveRecord::Base
  belongs_to :user
end
```

Get the user that wrote the first recipe

```
recipe = Recipe.find_by(id: 1)

recipe.user
```

```
SELECT  "users".* FROM "users" WHERE "users"."id" = 1
```

User & Recipe

```ruby
class User < ActiveRecord::Base
  has_many :recipes
end
```

```ruby
class Recipe < ActiveRecord::Base
  belongs_to :user
end
```

has_one

User & Credit Card

```ruby
class User < ActiveRecord::Base
  has_one :credit_card
end
```

```ruby
class CreditCard < ActiveRecord::Base
  belongs_to :user
end
```

The belongs_to side holds the reference

```
class CreateCreditCards < ActiveRecord::Migration
  def change
    create_table :credit_cards do |t|
      t.integer :user_id
      t.string :iban_number
      t.date :expiry_date

      t.timestamps null: false
    end
  end
end
```

Creating with associations

The old way of creating a recipe

```
Recipe.create(title: ‘Tasty Rubies',
content: ‘Boil 20 minutes and then …’,
user_id: 1)
```

The right way

```
user = User.find(1)

user.recipes.create(title: ‘Tasty Rubies',
content: ‘Boil 20 minutes and then …’)
```

has_many through

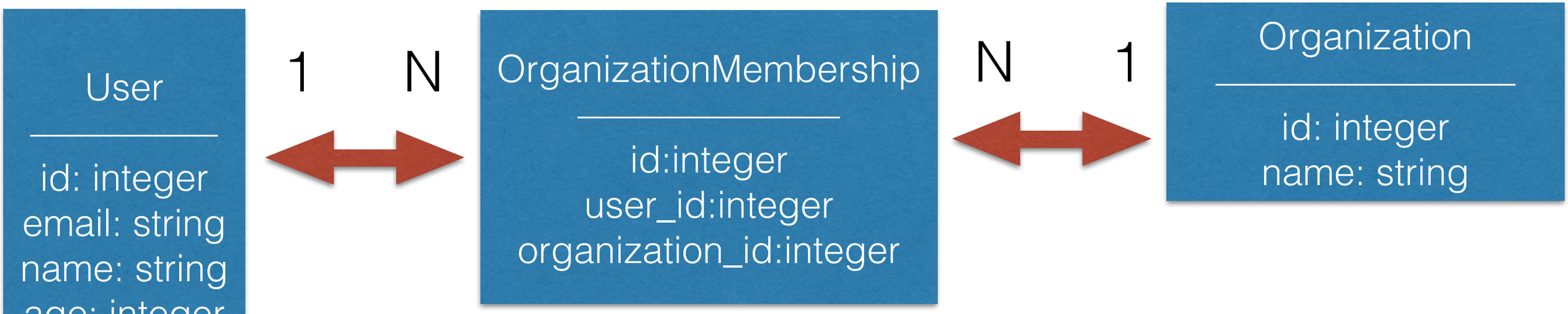
User & Organization

```ruby
class User < ActiveRecord::Base
end
```

```ruby
class Organization < ActiveRecord::Base
end
```

User & Organization

We need an intermediate table



User & Organization

```
class User < ActiveRecord::Base
end
```

```
class OrganizationMembership < ActiveRecord::Base
end
```

```
class Organization < ActiveRecord::Base
end
```

User & Organization

```
class OrganizationMembership < ActiveRecord::Base
  belongs_to :user
  belongs_to :organization
end
```

User & Organization

```ruby
class User < ActiveRecord::Base
  has_many :organization_memberships
end

class OrganizationMembership < ActiveRecord::Base
  belongs_to :user
  belongs_to :organization
end

class Organization < ActiveRecord::Base
  has_many :organization_memberships
end
```

```ruby
class User < ActiveRecord::Base
  has_many :organization_memberships
  has_many :organizations,
           through: :organization_memberships
end

class OrganizationMembership < ActiveRecord::Base
  belongs_to :user
  belongs_to :organization
end

class Organization < ActiveRecord::Base
  has_many :organization_memberships
  has_many :users,
           through: :organization_memberships
end
```

```ruby
class User < ActiveRecord::Base
  has_many :organization_memberships
  has_many :organizations,
           through: :organization_memberships
end
```

```ruby
user.organizations
```

```
SELECT "organizations".* FROM “organizations"
INNER JOIN “organization_memberships"
ON "organizations"."id" =
“organization_memberships"."organization_id"
WHERE "organization_memberships"."user_id" =  1
```