Prerequisites

1. RoR Installed and “Hello World” app from P0 working.
2. Understanding of the basics of Rails

Goals of this tutorial

Understand Active Record, the Object-Relational Mapping framework for Rails

Resources

- Active Record Basics: http://edgeguides.rubyonrails.org/active_record_basics.html
- Active Record Associations: http://guides.rubyonrails.org/association_basics.html
- Active Record Querying: http://guides.rubyonrails.org/active_record_querying.html
- Active Record Migrations: http://guides.rubyonrails.org/migrations.html
- Active Record Validations: http://edgeguides.rubyonrails.org/active_record_validations.html
**Topic 1: What is Active Record?**

Active Record is the M in MVC- the model - and is the ORM (Object-Relational Mapping) framework for Rails. It allows you to interact with the database without writing raw SQL queries, which is convenient and also prevents SQL-injection attacks.

**Object Relational Mapping**

Object-Relational Mapping is a technique that connects the rich objects of an application to tables in a relational database management system. Using ORM, the properties and relationships of the objects in an application can be easily stored and retrieved from a database without writing SQL statements directly and with less overall database access code.

Active Record gives us several mechanisms, the most important being the ability to:

- Represent models and their data.
- Represent associations between these models.
- Represent inheritance hierarchies through related models.
- Validate models before they get persisted to the database.
- Perform database operations in an object-oriented fashion.

**Topic 2: Active Record Objects**

Recall how to use a generator to create a model and the corresponding migration. This will create a User model, mapped to a users table at the database:

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

Now if you look at the model you will see:

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

Note that to create and Active Record model all you need to do is subclass ActiveRecord::Base

At this point, you have created the model, but the corresponding table in the database does not exist. By creating a database table you'll also have the ability to map the columns of each row in that table with the attributes of the instances of your model:
To create it, run the migration:

```
rake db:migrate
```

Now, if you look in `db/schema.rb`, you can see that your model is now part of the database schema:

```ruby
create_table "users", :force => true do |t|
  t.string   "name"
  t.integer  "age"
  t.datetime "created_at", :null => false
  t.datetime "updated_at", :null => false
end
```

A few notes about this:
1. Rails automatically creates and maintains timestamps (`created_at` and `updated_at`) for your models.
2. A primary key called `id` is also created for you automatically, although it is not listed in the schema.
3. The name of the table in the database is `users`, although the name of the model is `User`. Rails has pluralized your model name! If your model is called `Knife`, rails will actually create a table called `knives`.

You can revert your most recent migration by running:
```
rake db:rollback
```

**CRUD: Reading and Writing Data**

CRUD is an acronym for the four verbs we use to operate on data: Create, Read, Update and Delete. Active Record automatically creates methods to allow an application to read and manipulate data stored within its tables.

Active Record objects can be created from a hash, a block or have their attributes manually set after creation. The `new` method will return a new object while `create` will return the object and save it to the database.

`open up rails console to follow along with the code snippets`
user = User.create(name: ‘Cristina’, age: 22)

That was the Create part of the CRUD operations. But you can also Read: Now, you can fetch a particular user by id:

user = User.find(1)

Or by name:

user = User.where(name: ‘Cristina’).first

You can also chain the where method, which, contrary to intuition, will only cause one database access:

user = User.where(name: ‘Cristina’, age: 22).first

Or to get all users:

users = User.all

To check if a particular user exists, you can do this by id:

if User.exists?(1)
    # the user with id 1 exists
end

Or by something else:

if User.where(name: ‘Cristina’).exists?
    # a user named Cristina exists
end

Now for the updating. If you have an instance of a model and modify it, you can save it back to the database with the save method:

first_user = User.find_by(name: ‘Cristina’)
user.name = 'Cris'
user.save

Or you can use the update method which modifies the instance and saves it in one go:

first_user = User.find_by(name: ‘Cris’)
user.update(name: ‘Cristina’)
Finally, to delete a user from the database:

```ruby
User.find(1).destroy
```

**Validations**

Active Record allows you to validate the state of a model before it gets written into the database. There are several methods that you can use to check your models and validate that an attribute value is not empty, is unique and not already in the database, follows a specific format and many more.

Validation is a very important issue to consider when persisting to database, so the methods `create`, `save` and `update` take it into account when running: they return `false` when validation fails and they didn't actually perform any operation on database.

Change your model to be:

```ruby
class User < ActiveRecord::Base
  validates :name, presence: true
end
```

When you next run `create`, `update`, or `save` without defining a name, the `create` method will return `false`. You can use this to correctly display feedback to the user in the view such as errors on forms.

```ruby
User.create  # => false
```

**Topic 3: Associations**

Suppose we wanted to build a notifications system, where each user can have zero or more notifications, and each notification belongs to exactly one user. Given our `User` model above, you might create a `Notification` model like this:

```bash
rails generate model Notification user_id:integer message:string
```

Now if you have a notification and you want to get the user that it belongs to, you can look up the user by the `user_id` stored in the notification. If you want to find all the notifications that belong
to a particular user, you could use a “join” to do this, but there’s an easier way.

![Diagram of User and Notification relationships](image)

**Figure 1:** An example of a has_many/belongs_to association.

We can tell Rails about this relationship between notifications and users -- this is called an association. Open app/models/user.rb:

```ruby
class User < ActiveRecord::Base
  validates :name, presence: true
end
```

Now modify it so rails knows that users have notifications:

```ruby
class User < ActiveRecord::Base
  validates :name, presence: true
  has_many :notifications, dependent: :destroy
end
```

Similarly, open app/models/notification.rb:

```ruby
class Notification < ActiveRecord::Base
end
```
And modify it so rails knows that notifications belong to users:

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

Now it is easy to get all notifications for a particular user:

```ruby
# get all notifications that belong to the first user
notifications = User.first.notifications
```

Similarly, it is also easy to get the user that owns a particular notification:

```ruby
# get the user that owns the first notification
user = Notification.first.user
```

It is also easy to create a notification for a particular user:

```ruby
# give the first user a notification
User.first.notifications.create(message: "Hello, Cristina!")
```

The new notifications field of User uses some fancy Ruby metaprogramming to allow you to access the rows in the notifications table that belong to that particular user as if it were an ordinary array. By calling its create method, you are appending a new item to the array and also adding a new record to the database.

Recall that when we modified app/models/user.rb to tell rails that users have notifications, we had this clause:

```ruby
dependent: :destroy
```

This means that if we destroy a user, all of that user’s notifications are automatically destroyed as well:

```ruby
# destroy the first user and all his notifications
User.first.destroy
```

Note that there are several other types of associations, but has_many/belongs_to is perhaps the most common one. Rails supports six types of associations:

- **belongs_to**
• has_one
• has_many
• has_many :through
• has_one :through
• has_and_belongs_to_many

A :through association is used to indicate that the declaring model can be matched with instances of another model by proceeding through a third model. For example, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:

class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, through: :appointments
end

**Topic 4: Migrations**

Migrations are a feature of Active Record that allows you to evolve your database schema over time. Rather than write schema modifications in pure SQL, migrations allow you to use an easy Ruby DSL to describe changes to your tables.

You can think of each migration as being a new 'version' of the database. A schema starts off with nothing in it, and each migration modifies it to add or remove tables, columns, or entries. Active Record knows how to update your schema along this timeline, bringing it from whatever point it is in the history to the latest version. Active Record will also update your db/schema.rb file to match the up-to-date structure of your database.

Previously we have only run migrations that have been created when we use a generate command to make a model or scaffolding. You also need to generate migrations of your own to
make any changes to your models.

Suppose I now decide that I want to have Users have a nickname as well. I would need to generate a migration to add this new attribute to my database

```
rails generate migration AddNicknameToUser
```

Which creates an empty migration file that is appropriately names and looks like this:

```ruby
class AddNicknameToUser < ActiveRecord::Migration
  def change
    add_column :users, :nickname, :string
  end
end
```

In order to add a nickname, we need to add a column to the user database table. We can do that by changing the migration to have an `add_column` line

```ruby
class AddNicknameToUser < ActiveRecord::Migration
  def change
    add_column :users, :nickname, :string
  end
end
```

To run this migration and get your change into the database you would run `rake db:migrate` as per usual. Now, if you look in `db/schema.rb`, you can see that your changes have been reflected in the database schema.

If the migration name is of the form "AddXXXToYYY" or "RemoveXXXFromYYY" and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statements will be created. So we could have this type of migration created for us by running:

```
rails generate migration AddBirthdayToUser birthday:date
```

which creates automatically:

```ruby
class AddBirthdayToUser < ActiveRecord::Migration
  def change
    add_column :users, :birthday, :date
  end
end
```
Similarly, you can generate a migration to remove a column from the command line:

```ruby
rails generate migration RemoveBirthdayFromUsers birthday:date
```

which creates automatically:

```ruby
class RemoveBirthdayFromUsers < ActiveRecord::Migration
  def change
    remove_column :users, :birthday, :date
  end
end
```