# Lab 02: Ruby on Rails

**Instructions:** In this lab, we will get started with the Ruby programming language tools and the Ruby on Rails framework. After that, we will get a simple RoR application up and running.

## Get Apache and PostgreSQL up and running

Last time we installed Ubuntu 16.04, Apache and PostgreSQL. Before we begin the RoR part of this lab:

- If you haven't already, follow the Apache installation instructions in the LAP companion on the course Web site.
- If you haven't already, follow the PostgreSQL installation instructions in the LAP companion on the course Web site.

## Get Rails up and running

Now that we've got Linux, Apache, and PostgreSQL running, let's get a Rails app running with PostgreSQL as the underlying database engine. Visit Matt's Rails Companion on the course web site (`https://cis.ait.asia/course_offerings/440/rails-companion`) and follow the instructions to get the Ruby tools and Rails installed. Get an empty Rails app to run and connect to PostgreSQL.

## Quick Rails tutorial

We will be using Rails 5.0.0.1 Our simple example app will create database of students and projects:

1. Run `rails new studentdb -d postgresql`. It will run `bundle install` by default to find and install any gem dependencies.
2. Browse to the application's folder: `cd studentdb`
3. Run `bundle install` directly at the command line.
4. Uncomment this line in the file `Gemfile` to set up the Javascript runtime

   ```
   gem 'therubyracer', :platforms => :ruby
   ```

OK, now, let us look at the application structure and understand it.

1. `rake db:create` to automatically create the databases we need for the application. Check that the databases are actually created using `psql -l`.
2. Automatically generate a scaffold model/view/controller structure with Active Record ORM for two classes, `Student` and `Project`. First create a "Project" resource scaffolding:

   ```
   $ rails generate scaffold project name:string url:string
   ```

3. Go to folder `db/migrate/*` and try to understand what they do.
4. `rake db:migrate` to execute the database migrations.

5. `rails server`

6. Go to `http://localhost:3000/projects` to test. We can add/edit/delete projects.

7. You can easily create list of the projects in XML format. Open the file `app/controllers/projects_controller.rb` and alter the `index` action:

```
def index
 @projects = Project.all
 respond_to do |format|
   format.html
   format.json { render json: @projects }
   format.xml { render xml: @projects }
 end
end
```

   Now if you browse to `http://localhost:3000/projects.xml`, you will see all of the projects in XML format. Similarly for JSON format.

8. Now let us add `Student` class into the application. To create the new model, run this command:

```
$ rails generate model student studentid:string name:string project:references
```

9. Open the file `app/models/student.rb`.

```
class Student < ActiveRecord::Base
   belongs_to :project
end
```

   Active Record association `belongs_to :project` has been automatically added.

10. Open the database migration file `db/migrate/*`. You can see `t.references` line set up a foreign key column for the association between the two models, Student and Project.

11. `rake db:migrate`.

12. Edit the file `app/models/project.rb` to add the other side of association.

```
has_many :students
```

13. Open `config/routes.rb` and edit it as follows (near the top):

```
resources :projects do
 resources :students
end
```

14. Generate the controller for the `Student` model. Run this command:

```
$ rails generate controller students
```

   This will create the controller class `app/controllers/students_controller.rb` and the view directory `app/views/students/`.

15. Next we'll edit some files to enable adding students to a project. In `app/views/projects/show.html.erb`, add

```
<h2>Students</h2>
<table>
  <thead>
    <tr>
      <th>Student ID </th>
      <th>Name</th>
    </tr>
  </thead>
  <tbody>
    <% @project.students.each do |student| %>
      <tr>
        <td><%= student.studentid %></td>
        <td><%= student.name %></td>
      </tr>
    <% end %>
  </tbody>
</table>
<br/>
<h2>Add a student:</h2>
<%= form_for([@project, @project.students.build]) do |f| %>
  <div class="field">
    <%= f.label :studentid %><br />
    <%= f.text_field :studentid %>
  </div>
  <div class="field">
    <%= f.label :name %><br />
    <%= f.text_field :name %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
<br />
```

Then, in `app/controllers/students_controller.rb`, add the method

```
before_action :set_project

def create
  @student = @project.students.new(student_params)
  respond_to do |format|
    if @student.save
      format.html { redirect_to @project, notice: 'Student was successfully created.' }
    end
  end
end

private
  def set_project
    @project = Project.find(params[:project_id])
  end

  def student_params
    params.require(:student).permit(:studentid, :name)
  end
```

16. Finally, let's change the default route to be the project list: edit `config/routes.rb` and add

    ```
    root "projects#index"
    ```

    You can always check your routes with `rake routes`. Now visiting `http://localhost:3000` should take you right to your project list.

17. The next steps in a real agile Web development project might be to add unit, functional, and integration tests, and to *refactor* the code as necessary (the goal is Don't Repeat Yourself, or DRY). Look at the post "3 Rails refactoring tools" by Aaron Sumner (`http://everydayrails.com/2010/09/27/rails-refactoring-tools.html`)

## Active Record object-relational mapping in Rails

Now we'll discuss the object-relational mapping (ORM) provided by the Rails Active Record. The Active Record in Rails is a sophisticated ORM incorporating automatic synchronization with the database schema and transparent optimistic locking using row versioning. In most cases the database schema is read to define the attributes of each class of object in the OO schema.

Rails can (almost) automatically convert foreign key references in the database into object mappings. It supports one-to-one, one-to-many, and many-to-many mappings.

Just looking at the foreign key references isn't enough, however. It's impossible to tell whether a foreign key reference in table B to table A implies a one-to-one (or null) mapping between rows of B and A, or a one-to-many mapping from A to B.

First, Rails likes us to use some naming conventions for foreign key references. The reference field should be the class of the referred-to entity with `_id` appended. We already have this structure; it was done automatically for the foreign key reference from `students` to `projects` when we generated our `Student` model with the `rails generate` parameter `project:references`. See the method call `t.references :project` in your migration to create the `students` table. You can also set these things up manually.

For many-to-many relationships, in a relational database we use a join table containing foreign key reference columns for each of the tables we map between. Rails likes a join table to be named after the two tables it joins, with `_` in between, in alphabetical order. So if students could be part of multiple projects, we would need a join table named `projects_students`.

To specify one-to-one, one-to-many, and many-to-many relationships in Rails, we use the `has_one` (for the "one" side of a one-to-zero-or-none relationship), `has_many`, `belongs_to` (for the many side of a one-to-many relationship and the zero-or-none side of a one-to-zero-or-none relationship), and `has_and_belongs_to_many` (for both sides of a many-to-many relationship) Active Record methods to declare the associations. We also have `has_many :through` (for many-to-many connection with another model) and `has_one :through` (for one-to-one connection with another model).

We already used `belongs_to` and `has_many` to implement the many-to-one relationship between students and projects in the tutorial.

Next, we'll see how these concepts are applied in our little student/project database example.

1. Let's play with some Ruby objects of class Active Record and commit changes to the database. First, use the active scaffold you already built in the tutorial to populate `students` and `projects` with a few rows.

2. Now fire up the Rails console. This gives you direct access to the persistent objects in `studentdb`:

    ```
    % rails console
    >> Student.all
    => ...
    ```

    you'll see a list of all the available student objects. Nice feature of Rails 5.0.0.1 is that it shows log directly in console.

3. Select a student to play with, for example the student whose `id` field is 1:

```
>> s = Student.find(1)
=> ...
>> Project.all
=> ...
>> s.project = Project.find(2)
=> ...
>> s.save
=> true
```

Verify that the update propagates to the database and to the Web interface. Note that other processes will not see the modifications until you execute the `save` method to propagate the modifications to the database.

4. Play with the data some more:

```
>> s = Student.new :studentid => 123457, :name => "Moo Ping"
=> ...
>> s.save
=> false
>> p = Project.find_by_name "______"
=> ...
>> p.students.push s
=> ...
```

This example shows the beauty of *reflective* programming in the Rails framework. Rails examines our database schema then associates customized finder methods with our persistent objects *on the fly*.

That's it for today! Now you are beginning to feel your way around a Rails application, make simple changes to views, and understand Rails ORM concepts and utilization. Now you should start working on homework 1!

HAPPY CODING !!!