Models and Migrations Project 4

CS 142 Section – October 18, 2010

Overview

- ActiveRecord and Models
- Model Associations
- Migrations

ActiveRecord and Models

ActiveRecord: a Rails library that implements Object Relational Mapping (ORM)

What this means: you can easily translate information between Ruby objects and database values

Rails makes it easy to set up a database: instead of using SQL, you use models and migrations

In a nutshell:
Model class = database table
Individual Model objects = rows in that table

Models

To create a Rails model, use the command

```
>> ruby script/generate model <my_model>
```

This creates a bunch of files, but there are two that are important to us

- The actual model: app/models/my_model.rb
- A migration file in db/migrate/ (more on this later)

Models

Every model is a subclass of the ActiveRecord::Base class

We usually don’t specify model attributes directly in the model class file

Instead, ActiveRecord inspects the database schema and comes up with these attributes for us

Models: example from class

ActiveRecord reads our schema to configure our models dynamically

If we have the following database table named “students”:

id	name	birth	gpa	grad
1	Anderson	1987-10-22	3.9	2009
2	Jones	1990-04-16	2.4	2012
…	…	…	…	…

the Student model object will automatically have fields for id, name, birth, gpa, and grad

The CRUD methods (Create, Read, Update, Delete) allow us to access database rows as if they were objects

Brief review of CRUD

Use CRUD methods on objects to perform operations on the underlying database

Note: CRUD are names for basic DB operations, they are not necessarily the names of methods in Active Record

For example, to read a row from the students database, we call Students.find (…), not Students.read

CRUD: Create

We can create objects a couple different ways:

Constructor-style, passing parameters to create:

```
student = Student.create (:name => "Smith" …)
```

Plain creation, assigning attributes later:

```
student = Student.create
student.name = "Smith"
student.save                    # saves student to DB
```

Note: the create method automatically calls save, but in the second example, we alter the object after creation, so we need to call save again

Digression: save/create

There are actually two versions each of save and create

save – returns true if save was successful, nil otherwise (it might not if your model does validation)

save! – returns true if save was successful, raises exception otherwise

create – returns Active Record object regardless of whether data was actually saved

create! – returns Active Record object if data is valid, raises exception otherwise

Tradeoffs:

- if you don't use the ! (bang) methods, your saves can fail silently
- uncaught exceptions throw up generic crash pages for users

CRUD: Read

To read objects, use the find method:

```
# returns student with ID = 2
Student.find(2)

# returns first student with name “Lee”
Student.find_by_name(“Lee”)

# returns first student with gpa >= 3.0
Student.find(:first, :conditions => “gpa >= 3.0”)

# returns all students with gpa >= 3.0, sorted
Student.find(:all, :conditions => “gpa >= 3.0”, :order
   => “gpa DESC”);
```

There are a ton of optional parameters you can pass to find (see Section 18.5 of the Rails book)

CRUD: Update

Updating is nothing special: just find an object, update its attributes, and save it

```
Student.find(2)
student.name = “Scott”
student.gpa = 0.4
student.save
```

update_attributes offers a shorter call to accomplish the same thing by taking a hash of attributes to update

```
Student.find(2)
student.update_attributes(:name => “Scott”,
                              :gpa => 0.4)
# update_attributes automatically calls save
```

CRUD: Delete

There are two forms of deletion: database-level and row-level

Database-level: delete the object in the database that matches something we’ve specified

```
# deletes student w/ ID = 2
Student.delete(2)

# deletes students w/ IDs in the array [1,2,3]
Student.delete([1,2,3])

# self-explanatory, deletes all students
Student.delete_all
```

Row-level: after we’ve found an object, delete it

```
student = Student.find(2)
student.destroy
```

Again, section 18.5 describes all of this in greater detail if you need

Model Associations

Model associations allow us to express relationships between our database tables

They also make many operations more convenient

For example, if we have a students table and an advisors table, we would prefer to get a student’s advisor by saying `student.advisor`, rather than having to manually join the database tables

Model Associations

One-to-one: has_one and belongs_to
A student has one transcript.
A transcript belongs to one student.

```
class Student < ActiveRecord::Base
   has_one :transcript
end

class Transcript < ActiveRecord::Base
   belongs_to :student
end
```

Now, we can use `student.transcript` or `transcript.student`.

Model Associations

One-to-many: has_many and belongs_to
A student belongs to one advisor.
An advisor has many students.

```
class Student < ActiveRecord::Base
   has_one :transcript
   belongs_to :advisor
end

class Advisor < ActiveRecord::Base
   has_many :students
end
```

Model Associations

Many-to-many: has_and_belongs_to_many
A student has many courses, and a course has many students.

```
class Student < ActiveRecord::Base
   has_one :transcript
   belongs_to :advisor
   has_and_belongs_to_many :courses
end

class Course < ActiveRecord::Base
   has_and_belongs_to_many :students
end
```

Now, `student.courses` is an array of the courses that the student is taking, and `course.students` is an array of the students in a course.

Migrations

We’ve been talking about how models get their attributes from database schemas

How do we create these schemas? Migrations!

Migrations are classes containing code for updating database schemas

They’re reversible, and have timestamps embedded in their file names upon creation

Migrations are located in db/migrate/, and may be created either as a part of ruby script/generate model or separately via ruby script/generate migration

Migrations: Creating a table

Generally just two methods, self.up and self.down

- up = code for updating the DB
- down = code to undo that update

The Student model links up with the “students” table
So, the Student’s attributes will just be the columns added in this migration

```
class CreateStudents < ActiveRecord::Migration
  def self.up
     # make table called “students” with columns
     create_table :students do |t|
          t.column :name,       :string        # name
          t.column :birth,      :date          # birthdate
          t.column :gpa,        :float         # gpa
          t.column :grad,       :integer       # grad year
     end
  end

  def self.down
     # delete the table, reversing the changes
     drop_table :students
  end
end
```

Migrations: Data migrations

A data migration allows you to load data into the DB.
The sample migration below loads a couple of students.

```
class LoadStudents < ActiveRecord::Migration
  def self.up
      down  # calls self.down to remove students first
      student1 = Student.create(…)
      student2 = Student.create(…)
      student2.name = “James”
      student2.save
  end

  def self.down
      Student.delete_all # removes all students
  end
end
```

Running migrations

To run all migrations up to the most recent one:
>> rake db:migrate

To run migrations up to a certain one (must know timestamp):
>> rake db:migrate VERSION=20091019141500

Note: if the current version of the schema is newer than the migration you want to run, this command will undo every migration newer than the one specified

To undo all migrations (i.e. go back to the start):
>> rake db:migrate VERSION=0