Rapid Application Development

COSC2675 2017 Week 8

Dr. Andy Song
andy.song@rmit.edu.au

Last Week

- Multiple tables
- Association methods
- Dependent tables
- Nested resources
- Implementing nested controllers and views
- Many-to-many
- Join tables
- Many-to-many controllers
- Many-to-many views

Migrations

- To track changes to the database.
- Are effectively lists of changes. Each migration is a set of instructions on previous migration.
- To separate programming with DB
- Are reversible (almost for all operations)
- Are independent from database (relational or NoSQL)
- Support different databases at development, testing and deployment.
- Warning: you may have hard time if what migrations' db information is different from what is actually there.

Migration Basics

- Maintained in folder `db/migrate`
- Each file contains a set of changes.
- Migration is not automatic, needs `rails db:migrate`
- The `generate` command can create basic migration files
  `rails generate migration empty_migration`
- A file `[Time Stamp]_empty_migration.rb` is created.

```
class EmptyMigration < ActiveRecord::Migration
  def change
  end
end
```

- Change is the heart of migration. Roll back will undo the changes.

Migration Operations

- `add_column`
- `add_foreign_key`
- `add_index`
- `add_reference`
- `add_timestamps`
- `change_column_default` (must supply :from and :to)
- `change_column_null`
- `create_join_table`
- `create_table`
- `disable_extension`
- `drop_join_table`
- `drop_table` (must supply a block)
- `enable_extension`

Migration Operations …

- `remove_column` (must supply a type)
- `remove_foreign_key` (must supply a second table)
- `remove_index`
- `remove_reference`
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `rename_table`

If want to go beyond the above operations, then you need to use `self.up` and `self.down` instead of change.

Warning on unsaved files: run the migration but didn't save the file, then save the file and roll back… It might be difficult to fix the problem. Make sure you've saved all files before migration.

Running Migrations

- `rails db:migrate`

To update your database. It updates folder `db/migrate` and file `db/schema.rb`

If forgot to run it, you might see lots of missing or nil object errors.

- `rails db:rollback`

To remove the last migration. It also updates `db/schema.rb`

To remove multiple migrations: `rails db:rollback STEP=n`

Warning: when delete a column or table, the data is also lost.

Running Migrations

- `rails db:drop`

This is throw away all the migrations and all data!

- `rails db:reset`

This is different with drop. It obliterates the database and then builds a new one using `db/schema.rb`, using the last structure created.

- `rails db:create`

Create a new database without requiring you to know the internal details.

NOTE: One migration can operate multiple tables. For beginners, it is a good idea to start from operating only on a single table in each migration.

Inside Migrations

```
> rails generate scaffold student given_name:string
middle_name:string family_name:string date_of_birth:date
grade_point_average:decimal start_date:date
```

This command generates many files, including the migration file
`db/migrate/[Time Stamp]_create_students.rb`

```
class CreateStudents < ActiveRecord::Migration
  def change
    create_table :students do | t |
      t.string :given_name
      t.string :middle_name
      t.string :family_name
      t.date :date_of_birth
      t.decimal :grade_point_average
      t.date :start_date t.timestamps
    end
  end
end
```


Working with Tables

By convention, migrations that create new tables start with "create"

```
> rails generate migration CreateBook
```

This produces migration file
`db/migrate/[Time Stamp]_create_book.rb`

```
class CreateBook < ActiveRecord::Migration
  def change
    create_table :books do | t |
    end
  end
end
```

Columns then can be added without the `create_table` method.

Note a column `id` will be automatically added unless

```
create_table :books id:false do |t|
```

Working with Tables…

```
> rails rails generate migration CreateBookWithColumns
title:string author:string isbn:integer price:float
published_date:date
```

This command generates many files, including the migration file `db/migrate/[Time Stamp]_create_book_with_columns.rb`

```
class CreateBookWithColumns < ActiveRecord::Migration
  def change
    create_table :book_with_columns do | t |
      t.string :title
      t.string :author
      t.integer :isbn
      t.float :price
      t.date :published_date
    end
  end
end
```


Supported Data Types (11 types)

- `:string`
- `:text`
- `:integer`
- `:float`
- `:decimal`
- `:datetime`
- `:timestamp`
- `:time`
- `:date`
- `:binary`
- `:boolean`

Note: `timestamp` is not `timestamps`. The latter is a method to manage creation/modification times (`created_at`, `updated_at`).

Parameters of Data Types (11 types)

All data types accept these two named parameters

- `default: value`
  Note: users won't see them. They will be overwritten by user input even if it is nothing.
- `null: true|false` (whether a null value is acceptable)

String, text, binary and integer accept

- `limit: size` (the permitted length in characters or bytes)

Decimal type also accepts

- `precision: value` (how many digits the number can have)
- `scale: value` (how many digits appear after the decimal)

Warning: always specify precision and scale if your application might move across different databases to minimize surprises. Different DB treat decimal differently.

Example of Parameters

```
class CreateBookWithColumns < ActiveRecord::Migration
  def change
    create_table :book_with_columns do | t |
      # entries will be limited to 100 characters and
      # cannot be left empty
      t.string :title, limit: 100
      # entries will be limited to 45 characters
      t.string :author, limit: 45
      # entries will be limited to 13 characters
      t.integer :isbn, limit: 13
      # entries will follow the format "xxxx.xx"
      t.decimal :price, precision: 6, scale: 2
      # this sets the default value to today's date.
      # However, this is not automatically persisted
      # to the model and will be overwritten with
      # user-entered data.
      t.date :published_date, default: Date.today
    end
  end
end
```

Remarks on parameters

Parameters add constraints on data. However it will be easier to implement them as data validation.

Precision and scale are recommended for decimal types due to database incompabilities.

Custom types can be created if really necessary.

Set `config.active_record.schema_format` which is inside of `config/environment.rb` file.

In general, you shouldn't do that.

Add Columns

Rails allows you modify the columns easily so new ideas can be quickly implemented.

```
> rails g migration AddLanguageToBooks language:string
```

This produces migration file
`db/migrate/[Time Stamp]_add_language_to_book.rb`

```
class AddLanguageToBook < ActiveRecord::Migration
  def change
    add_column :books, :language, :string
  end
end
```

This `add_column` method takes three parameters. The table, the name for the new column and the data type of the new column.

Parameters discussed early are allowed.

Delete Columns

Removing a column is also often needed.

```
> rails g migration RemovePublished_dateFromBooks
published_date:date
```

This produces migration file
`db/migrate/[Time Stamp]_remove_published_date_from_books.rb`

```
class RemovePublishedDateFromBooks
      < ActiveRecord::Migration
  def change
    remove_column :books, :published_date, :date
  end
end
```

Note, methods `add_columns` and `remove_columns` are available as well. They support the alternation of multiple columns.

Modify Columns

Change a column is also often needed .

```
> rails g migration alter_column_books_price
```

This produces migration file
`db/migrate/[Time Stamp]_alter_column_books_price.rb`

```
class AlterColumnBooksPrice < ActiveRecord::Migration
  def change
  end
end
```

Note (1) the migration command (2) the empty change method.

A change of column is IRREVERSIBLE!

So we need to provide `up` and `down` for method `change_column.`

Modify Columns…

Manually change the migration file:

`db/migrate/[Time Stamp]_alter_column_books_price.rb`

```
class AlterColumnBooksPrice < ActiveRecord::Migration

  def self.up
    # Change the price column to accept 7 digits
    change_column :books, :price, precision: 7, scale: 2
  end

  def self.down
    #Revert the price column back to 6 digits
    change_column :books, :price, precision: 6, scale: 2
  end
End
```

Save the file before running `rails db:migrate`

Indices

- Indices can be viewed as a data structure to organize data so finding target data can be quicker.
- Indices improve "reading" performance, but slow down "writing" performance, as indices need to be updated when new data is entered.
- By default, Rails use the id column to build the index.
- If other columns will be searched regularly, then change the index to these columns
- Use methods `add_index` and `remove_index`

Add Indices

In the book example, we can index on ISBN.

```
class CreateBookWithColumns < ActiveRecord::Migration
  def change
    create_table :book_with_columns do | t |
      t.string :title
      t.string :author
      t.integer :isbn
      t.float :price
      t.date :published_date
    end

    add_index :book_with_columns, :isbn
  end
end
```

The `add_index` method can be added to virtually any migration, either standalone or mixed with other operations.

Add Indices

Another way to add index to append a index modifier to a column definition.

```
class AddLanguageToBook < ActiveRecord::Migration
  def change
    add_column :books, :language, :string, index: true
  end
end
```

NOTE: indices should always be added after the creation of the table or the column. Otherwise, rollback will go wrong.

Migration Methods

See API `ActiveRecord::ConnectionAdapters::SchemaStatements` where you can find most of the migration methods, including

```
add_belongs_to                  add_column
add_foreign_key                 add_index
add_index_options               add_index_sort_order
add_reference                   add_timestamps
assume_migrated_upto_version    change_column
change_column_default           change_column_null
change_table                    column_exists?
columns                         columns_for_distinct
create_alter_table              create_join_table
create_table                    create_table_definition
drop_join_table                 dump_schema_information
foreign_key_column_for          foreign_key_name
foreign_keys                    index_exists? index_name
index_name_exists?              index_name_for_remove
```


Migration Methods…

```
index_name_exists?              index_name_for_remove
initialize_schema_migrations_table
native_database_types           options_include_default?
quoted_columns_for_index        remove_belongs_to
remove_column                   remove_columns
remove_foreign_key              remove_index
remove_index!                   remove_reference
remove_timestamps               rename_column
rename_column_indexes           rename_index
rename_table                    rename_table_indexes
table_alias_for                 table_exists?
type_to_sql                     update_table_definition
validate_index_length!
```

These listed methods offer a wide of functionalities although some of them are seldom used.

Rails documentation, Rails API and APIdock are good references.

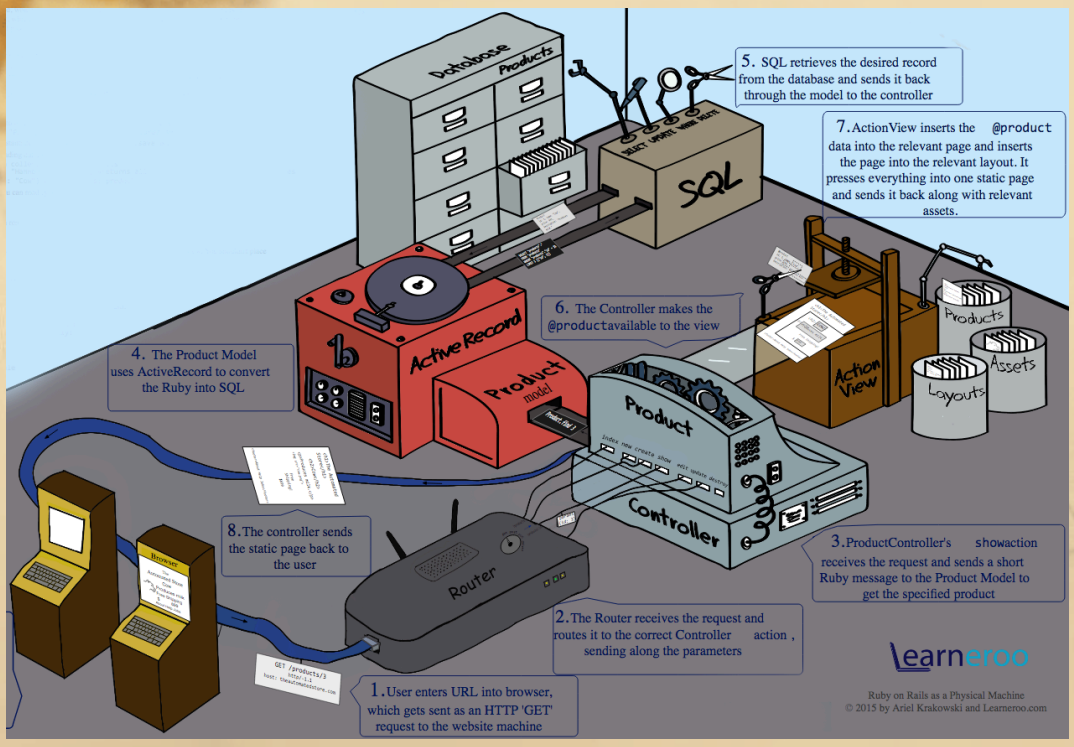
SQL execute method

- In `ActiveRecord::ConnectionAdapters` there is a relevant method `execute(sql, name=nil)`
- This allows you directly use SQL statements to manipulate your DB.
- Note that execute is not reversible.
- You must explicitly define `self.up` and `self.down` to allow modification and rolling back
- This practice violates Rails' convention over configuration philosophy.
- Almost any SQL statements can be accomplished using Rails migration methods.

MVC

- Model-View-Controller, a software architecture which divides an application into three interconnected parts.
- MVC facilitates efficient code reuse and parallel development
- It was introduced into Smalltalk in the 1970s.
- High cohesion: logical grouping of related actions
- Low coupling: separation among models, views or controllers
- Models can have multiple views
- Easy to modify
- Navigation through code can be complex
- Requirement on consistency at multiple places
- Steep learning curve, especially when multiple technologies are involved.

MVC in Rails



(Ariel Krakowski's example)

MVC in Rails - Model

We can see the MVC structure in Rails' structure to organize files.

Models inherit from ActiveRecord, an Object Relational Mapping (ORM) framework, treating database entries as Ruby objects.

Code that relates to data should be in the model.

```
rails g model Product string:name description:text
rails g migration AddPriceToProducts price:integer
rails db:migrate
```

CRUD Create -- `Product.create(name:"X", description: "Y")`

Read -- `product = Product.find_by(name:"X")`

Update -- `product.description = "Z"`

Delete -- `product.destroy`

(Ariel Krakowski's example)

MVC in Rails - Controller

A controller controls access to an application and makes the data from the model available to the view. You shouldn't place too much code in the Controller; keep a "fat model, skinny controller".

```
rails g controller Product index
```

Controllers can access GET/POST parameters from params hash:

```
class ProductsController < ApplicationController
  def index
    if params[:status] == "sale"
      @products = Product.on_sale
    else
      @products = Product.all
    end
  end
end
```

(Ariel Krakowski's example)

MVC in Rails - View

Erb pages will be processed and presented to viewers. Views contain Ruby code to display data from the controller. The actual logic and database operations are not in the views.

`<%  %>` Execute code without returning anything

`<%= %>` Execute code and display its output

Example: to loops through products and display all product names.

```
<h1> All Products </h1>
<% @products.each do |product| %>
  <%= product.name %> <br>
<% end %>
```

(Ariel Krakowski's example)

Summary

- Migration
- Migration Basics
- Migration Operations
- Working with Tables
- Data Types and Parameters
- Working with Columns
- Indices
- MVC
- MVC in Rails