Rails Migration Guide

by Prof. Houda Bouamor
Information Systems, CMU-Q

Why would I need it?

Rails migrations are useful any time you need to make a change to your application's database. If you're working with Rails Active Records, manipulating the database directly is a bad idea.

Rails Migration allows you to use Ruby to define changes to your database schema, making it possible to use a version control system to keep things synchronized with the actual code. Using migrations, the database schema is updated and stored in the schema.rb file (in the db/ folder). A database schema is the collection of table schemas for a whole database. A table schema is the logical definition of a table - it defines what the name of the table is, and what the name and type of each column is.

As we saw in the example covered in the class (the Owner model and owners table) Active Record uses migrations to update your application database and add a new table (Owner).

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.

What Can Rails Migration Do?

You can use migrations to make any changes you need to the database your application connects to.

You can also roll migrations back, assuming it didn't do something irreversible like destroying data.

Let's take a look at six specific operations:

1. Add a new table (associated to a model)

Adding a new model to an application is a frequent change. When you add a new model to a Rails application, it generates a migration to create the corresponding table for you. If that's not what you need, you can write your own or edit the current migration by editing the corresponding migration file (*YYYYMMDDHHMMSScreatetable.rb*).

2. Add a column

Another common change in an application is adding a field to an existing object. So adding a column to a database table is a typical migration. If you're working with Active Records, Rails will create the migration for you.

In order to add a new column to the table (attribute to the model object), you should create a standalone migration independantly from the model itself.

If the migration name is of the form "AddColumnToTable" and is followed by a list of column names and types then a migration containing the appropriate add_column statements will be created.

```
rails generate migration AddColumnToTable column1:integer
```

This will generate a migration file in this format
`YYYYMMDDHHMMSS_add_column_to_table.rb`

The name of the migration class (CamelCased version) should match the latter part of the file name. For example 20210216120000_create_owners.rb should define the class CreateOwners (the migration) and 20210216130000_add_age_to_owners.rb should define AddAgeToOwners.

Rails uses the timestamp to determine which migration should be run and in what order, so if you're copying a migration from another application or generate a file yourself, be aware of its position in the order.

We can also add more than one columns. For example:

```
rails generate migration AddDetailsToOwners age:integer ssn:string
```

This generates the following migration:

```
class AddDetailsToOwners < ActiveRecord::Migration[5.2]
    def change
        add_column :owners, :age, :integer
        add_column :onwers, :ssn, :string
    end
end
```

3. Remove a column

If the migration name is of the form "RemoveColumnFromTable" and is followed by a list of column names and types then a migration containing the appropriate remove_column statements will be created.

```
rails generate migration RemoveColumnToTable column:string
```

4. Drop a table

In rails, a table is not dropped using a migration. A table is dropped from the database automatically and the migration file associated with it is also removed, when the corresponding model is destroyed.

To destroy a model, we use the following command:

```
rails destroy model ModelName
```

For example, to destroy the Owner model, we can excute the following:

```
rails destroy model Owner
```

This generates the following:

```
Running via Spring preloader in process 79492
  invoke  active_record
  remove    db/migrate/20210216075606_create_owners.rb
  remove    app/models/owner.rb
  invoke    test_unit
  remove      test/models/owner_test.rb
  remove      test/fixtures/owners.yml
```

These first four ways of using migrations are far and away the most popular, but there are two lesser-known (but still very useful) ways to utlize migrations in Rails.

5. Add data to a table

While most times we can use `rake db:seed` or create our own, more extensive populating scripts to add content to a database (in PATS we have `rake db:populate` and `rake db:contexts`), the process of adding data to a database can also be accomplished via migrations. One use case for doing it this way might be if you want to always create a default user at the time the database is created so you know there will always be one person who can log in and access the system.

An example of this in PATS would be the following migration:

```
class AddDefaultUser < ActiveRecord::Migration[5.2]
    def change
        # create a new instance of User and populate
        vet = User.new
        vet.username = "vet"
        vet.email = "vet@example.com"
        vet.password = "yodel"
        vet.password_confirmation = "yodel"
        # save with bang will throw exception on failure
        vet.save!
    end
end
```

Again, beyond this use case there are often better options for adding data to our database, but it is available if needed.

6. Add functions and triggers to the database

There may be times when we need to interact with the database directly with SQL -- such as adding an extension, creating a function, or applying a trigger -- and we want to make sure that the modifications we've performed on the database are preserved and that other developers in the future will have these modifications. It is possible to execute raw SQL within a migration in order to achieve this.

An example of this in PATS would be the final segment of the class when we wish to add phonetic ("sounds like") searching to our application and find other records that are very similar in how they sound. Postgres has a great set of tools for this, one of which is `dmetaphone` and is available in the postgres extension `fuzzystrmatch`. For us to be able to add this functionality to our app, we need to install that extension and then write a simple function within the database. We can achieve both with the following migration:

```
class AddPgSearchDmetaphoneSupportFunctions < ActiveRecord::Migration[5.2
]
    def change
        # add the extension first...
        say_with_time("Adding psql extensions if not existing:") do
            execute <<-'SQL'
            CREATE EXTENSION IF NOT EXISTS fuzzystrmatch;
            SQL
        end

        # now add the search function needed on the database side...
        say_with_time("Adding support functions for pg_search :dmetaphone")
            execute <<-'SQL'
            CREATE OR REPLACE FUNCTION pg_search_dmetaphone(text)
                RETURNS text LANGUAGE SQL IMMUTABLE STRICT AS $function$
                SELECT array_to_string(ARRAY(SELECT dmetaphone(unnest(regex
                $function$;
            SQL
        end
    end
end
```

Once this migration is created, all future developers will have this functionality available when they download the code and simply run `rails db:migrate`. Of course, the danger with this is that such a migration will ony run on Postgres; any other DBMS tries to use this and it will err. Writing raw SQL for db extensions and triggers might be necessary at times, but to the extent we can stick to writing code in Ruby and letting ActiveRecord handle conversion to SQL, the better off we generally are.