Active Record 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.

After reading this guide, you will know:

- **The generators you can use to create them.**
- **The methods Active Record provides to manipulate your database.**
- **The Rake tasks that manipulate migrations and your schema.**
- **How migrations relate to `schema.rb`.**

Chapters

1 Migration Overview

Migrations are a convenient way to alter your database schema over time in a consistent and easy way. They use a Ruby DSL so that you don't have to write SQL by hand, allowing your schema and changes to be database independent.

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.

Here's an example of a migration:

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

This migration adds a table called `products` with a string column called `name` and a text column called `description`. A primary key column called `id` will also be added implicitly, as it's the default primary key for all Active Record models. The `timestamps` macro adds two columns, `created_at` and `updated_at`. These special columns are automatically managed by Active Record if they exist.

Note that we define the change that we want to happen moving forward in time. Before this migration is run, there will be no table. After, the table will exist. Active Record knows how to reverse this migration as well: if we roll this migration back, it will remove the table.

On databases that support transactions with statements that change the schema, migrations are wrapped in a transaction. If the database does not support this then when a migration fails the parts of it that succeeded will not be rolled back. You will have to rollback the changes that were made by hand.

There are certain queries that can't run inside a transaction. If your adapter supports DDL transactions you can use `disable_ddl_transaction!` to disable them for a single migration.

If you wish for a migration to do something that Active Record doesn't know how to reverse, you can use `reversible`:

```
class ChangeProductsPrice < ActiveRecord::Migration
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

Alternatively, you can use `up` and `down` instead of `change`:

```
class ChangeProductsPrice < ActiveRecord::Migration
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

2 Creating a Migration

2.1 Creating a Standalone Migration

Migrations are stored as files in the `db/migrate` directory, one for each migration class. The name of the file is of the form `YYYYMMDDHHMMSS_create_products.rb`, that is to say a UTC timestamp identifying the migration followed by an underscore followed by the name of the migration. The name of the migration class (CamelCased version) should match the latter part of the file name. For example `20080906120000_create_products.rb` should define class `CreateProducts` and `20080906120001_add_details_to_products.rb` should define `AddDetailsToProducts`. Rails uses this 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.

Of course, calculating timestamps is no fun, so Active Record provides a generator to handle making it for you:

```
$ bin/rails generate migration AddPartNumberToProducts
```

This will create an empty but appropriately named migration:

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

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.

```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string
```

will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
  end
end
```

If you'd like to add an index on the new column, you can do that as well:

```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string:index
```

will generate

```
class AddPartNumberToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

Similarly, you can generate a migration to remove a column from the command line:

```
$ bin/rails generate migration RemovePartNumberFromProducts
part_number:string
```

generates

```
class RemovePartNumberFromProducts < ActiveRecord::Migration
  def change
    remove_column :products, :part_number, :string
  end
end
```

You are not limited to one magically generated column. For example:

```
$ bin/rails generate migration AddDetailsToProducts
part_number:string price:decimal
```

generates

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

If the migration name is of the form "CreateXXX" and is followed by a list of column names and types then a migration creating the table XXX with the columns listed will be generated. For example:

```
$ bin/rails generate migration CreateProducts name:string
part_number:string
```

generates

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

As always, what has been generated for you is just a starting point. You can add or remove from it as you see fit by editing the `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb` file.

Also, the generator accepts column type as `references`(also available as `belongs_to`). For instance:

```
$ bin/rails generate migration AddUserRefToProducts user:references
```

generates

```
class AddUserRefToProducts < ActiveRecord::Migration
  def change
    add_reference :products, :user, index: true
  end
end
```

This migration will create a `user_id` column and appropriate index.

There is also a generator which will produce join tables if `JoinTable` is part of the name:

```
$ bin/rails g migration CreateJoinTableCustomerProduct customer
product
```

will produce the following migration:

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

2.2 Model Generators

The model and scaffold generators will create migrations appropriate for adding a new model. This migration will already contain instructions for creating the relevant table. If you tell Rails what columns you want, then

statements for adding these columns will also be created. For example, running:

```
$ bin/rails generate model Product name:string description:text
```

will create a migration that looks like this

```
class CreateProducts < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

You can append as many column name/type pairs as you want.

2.3 Supported Type Modifiers

You can also specify some options just after the field type between curly braces. You can use the following modifiers:

- `limit` Sets the maximum size of the `string/text/binary/integer` fields.
- `precision` Defines the precision for the `decimal` fields, representing the total number of digits in the number.
- `scale` Defines the scale for the `decimal` fields, representing the number of digits after the decimal point.
- `polymorphic` Adds a `type` column for `belongs_to` associations.
- `null` Allows or disallows `NULL` values in the column.

For instance, running:

```
$ bin/rails generate migration AddDetailsToProducts
'price:decimal{5,2}' supplier:references{polymorphic}
```

will produce a migration that looks like this

```
class AddDetailsToProducts < ActiveRecord::Migration
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true, index:
true
  end
end
```

3 Writing a Migration

Once you have created your migration using one of the generators it's time to get to work!

3.1 Creating a Table

The `create_table` method is one of the most fundamental, but most of the time, will be generated for you from using a model or scaffold generator. A typical use would be

```
create_table :products do |t|
  t.string :name
end
```

which creates a `products` table with a column called `name` (and as discussed below, an implicit `id` column).

By default, `create_table` will create a primary key called `id`. You can change the name of the primary key with the `:primary_key` option (don't forget to update the corresponding model) or, if you don't want a primary key at all, you can pass the option `id: false`. If you need to pass database specific options you can place an SQL fragment in the `:options` option. For example:

```
create_table :products, options: "ENGINE=BLACKHOLE" do |t|
  t.string :name, null: false
end
```

will append `ENGINE=BLACKHOLE` to the SQL statement used to create the table (when using MySQL, the default is `ENGINE=InnoDB`).

3.2 Creating a Join Table

Migration method `create_join_table` creates a HABTM join table. A typical use would be:

```
create_join_table :products, :categories
```

which creates a `categories_products` table with two columns called `category_id` and `product_id`. These columns have the option `:null` set to `false` by default. This can be overridden by specifying the `:column_options` option.

```
create_join_table :products, :categories, column_options: {null:
true}
```

will create the `product_id` and `category_id` with the `:null` option as `true`.

You can pass the option `:table_name` when you want to customize the table name. For example:

```
create_join_table :products, :categories, table_name: :categorization
```

will create a `categorization` table.

`create_join_table` also accepts a block, which you can use to add indices (which are not created by default) or additional columns:

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

3.3 Changing Tables

A close cousin of `create_table` is `change_table`, used for changing existing tables. It is used in a similar fashion to `create_table` but the object yielded to the block knows more tricks. For example:

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

removes the `description` and `name` columns, creates a `part_number` string column and adds an index on it. Finally it renames the `upccode` column.

3.4 When Helpers aren't Enough

If the helpers provided by Active Record aren't enough you can use the `execute` method to execute arbitrary SQL:

```
Product.connection.execute('UPDATE `products` SET `price`=`free`
WHERE 1')
```

For more details and examples of individual methods, check the API documentation. In particular the documentation for `ActiveRecord::ConnectionAdapters::SchemaStatements` (which provides the methods available in the `change`, `up` and `down` methods), `ActiveRecord::ConnectionAdapters::TableDefinition` (which provides the methods available on the object yielded by `create_table`) and `ActiveRecord::ConnectionAdapters::Table` (which provides the methods available on the object yielded by `change_table`).

3.5 Using the `change` Method

The `change` method is the primary way of writing migrations. It works for the majority of cases, where Active Record knows how to reverse the migration automatically. Currently, the `change` method supports only these

migration definitions:

- `add_column`
- `add_index`
- `add_reference`
- `add_timestamps`
- `create_table`
- `create_join_table`
- `drop_table` (must supply a block)
- `drop_join_table` (must supply a block)
- `remove_timestamps`
- `rename_column`
- `rename_index`
- `remove_reference`
- `rename_table`

`change_table` is also reversible, as long as the block does not call `change`, `change_default` or `remove`.

If you're going to need to use any other methods, you should use `reversible` or write the `up` and `down` methods instead of using the `change` method.

3.6 Using `reversible`

Complex migrations may require processing that Active Record doesn't know how to reverse. You can use `reversible` to specify what to do when running a migration what else to do when reverting it. For example:

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :products do |t|
      t.references :category
    end

    reversible do |dir|
      dir.up do
        #add a foreign key
        execute <<-SQL
          ALTER TABLE products
            ADD CONSTRAINT fk_products_categories
            FOREIGN KEY (category_id)
            REFERENCES categories(id)
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE products
            DROP FOREIGN KEY fk_products_categories
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
end
```

Using `reversible` will ensure that the instructions are executed in the right order too. If the previous example migration is reverted, the `down` block will be run after the `home_page_url` column is removed and right before the table `products` is dropped.

Sometimes your migration will do something which is just plain irreversible; for example, it might destroy some data. In such cases, you can raise `ActiveRecord::IrreversibleMigration` in your `down` block. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

3.7 Using the `up`/`down` Methods

You can also use the old style of migration using `up` and `down` methods instead of the `change` method. The `up` method should describe the transformation you'd like to make to your schema, and the `down` method of your migration should revert the transformations done by the `up` method. In other words, the database schema should be unchanged if you do an `up` followed by a `down`. For example, if you create a table in the `up` method, you should drop it in the `down` method. It is wise to reverse the transformations in precisely the reverse order they were made in the `up` method. The example in the `reversible` section is equivalent to:

```
class ExampleMigration < ActiveRecord::Migration
  def up
    create_table :products do |t|
      t.references :category
    end

    # add a foreign key
    execute <<-SQL
      ALTER TABLE products
        ADD CONSTRAINT fk_products_categories
        FOREIGN KEY (category_id)
        REFERENCES categories(id)
    SQL

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE products
        DROP FOREIGN KEY fk_products_categories
    SQL

    drop_table :products
  end
end
```

If your migration is irreversible, you should raise `ActiveRecord::IrreversibleMigration` from your `down` method. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

3.8 Reverting Previous Migrations

You can use Active Record's ability to rollback migrations using the `revert` method:

```
require_relative '2012121212_example_migration'

class FixupExampleMigration < ActiveRecord::Migration
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```

The `revert` method also accepts a block of instructions to reverse. This could be useful to revert selected parts of previous migrations. For example, let's imagine that `ExampleMigration` is committed and it is later decided it would be best to serialize the product list instead. One could write:

```
class SerializeProductListMigration < ActiveRecord::Migration
  def change
    add_column :categories, :product_list

    reversible do |dir|
      dir.up do
        # transfer data from Products to Category#product_list
      end
      dir.down do
        # create Products from Category#product_list
      end
    end

    revert do
      # copy-pasted code from ExampleMigration
      create_table :products do |t|
        t.references :category
      end

      reversible do |dir|
        dir.up do
          #add a foreign key
          execute <<-SQL
            ALTER TABLE products
              ADD CONSTRAINT fk_products_categories
              FOREIGN KEY (category_id)
              REFERENCES categories(id)
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE products
              DROP FOREIGN KEY fk_products_categories
          SQL
        end
      end
```

```
      # The rest of the migration was ok
    end
  end
end
```

The same migration could also have been written without using `revert` but this would have involved a few more steps: reversing the order of `create_table` and `reversible`, replacing `create_table` by `drop_table`, and finally replacing `up` by `down` and vice-versa. This is all taken care of by `revert`.

4 Running Migrations

Rails provides a set of Rake tasks to run certain sets of migrations.

The very first migration related Rake task you will use will probably be `rake db:migrate`. In its most basic form it just runs the `change` or `up` method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.

Note that running the `db:migrate` task also invokes the `db:schema:dump` task, which will update your `db/schema.rb` file to match the structure of your database.

If you specify a target version, Active Record will run the required migrations (change, up, down) until it has reached the specified version. The version is the numerical prefix on the migration's filename. For example, to migrate to version 20080906120000 run:

```
$ bin/rake db:migrate VERSION=20080906120000
```

If version 20080906120000 is greater than the current version (i.e., it is migrating upwards), this will run the `change` (or `up`) method on all migrations up to and including 20080906120000, and will not execute any later migrations. If migrating downwards, this will run the `down` method on all the migrations down to, but not including, 20080906120000.

4.1 Rolling Back

A common task is to rollback the last migration. For example, if you made a mistake in it and wish to correct it. Rather than tracking down the version number associated with the previous migration you can run:

```
$ bin/rake db:rollback
```

This will rollback the latest migration, either by reverting the `change` method or by running the `down` method. If you need to undo several migrations you can provide a `STEP` parameter:

```
$ bin/rake db:rollback STEP=3
```

will revert the last 3 migrations.

The `db:migrate:redo` task is a shortcut for doing a rollback and then migrating back up again. As with the `db:rollback` task, you can use the `STEP` parameter if you need to go more than one version back, for example:

```
$ bin/rake db:migrate:redo STEP=3
```

Neither of these Rake tasks do anything you could not do with `db:migrate`. They are simply more convenient, since you do not need to explicitly specify the version to migrate to.

4.2 Setup the Database

The `rake db:setup` task will create the database, load the schema and initialize it with the seed data.

4.3 Resetting the Database

The `rake db:reset` task will drop the database and set it up again. This is functionally equivalent to `rake db:drop db:setup`.

This is not the same as running all the migrations. It will only use the contents of the current `schema.rb` file. If a migration can't be rolled back, `rake db:reset` may not help you. To find out more about dumping the schema see **Schema Dumping and You** section.

4.4 Running Specific Migrations

If you need to run a specific migration up or down, the `db:migrate:up` and `db:migrate:down` tasks will do that. Just specify the appropriate version and the corresponding migration will have its `change`, `up` or `down` method invoked, for example:

```
$ bin/rake db:migrate:up VERSION=20080906120000
```

will run the 20080906120000 migration by running the `change` method (or the `up` method). This task will first check whether the migration is already performed and will do nothing if Active Record believes that it has already been run.

4.5 Running Migrations in Different Environments

By default running `rake db:migrate` will run in the `development` environment. To run migrations against another environment you can specify it using the `RAILS_ENV` environment variable while running the command. For example to run migrations against the `test` environment you could run:

```
$ bin/rake db:migrate RAILS_ENV=test
```

4.6 Changing the Output of Running Migrations

By default migrations tell you exactly what they're doing and how long it took. A migration creating a table

and adding an index might produce output like this

```
==  CreateProducts: migrating
=============================================
-- create_table(:products)
   -> 0.0028s
==  CreateProducts: migrated (0.0028s)
=======================================
```

Several methods are provided in migrations that allow you to control all this:

Method	Purpose
suppress_messages	Takes a block as an argument and suppresses any output generated by the block.
say	Takes a message argument and outputs it as is. A second boolean argument can be passed to specify whether to indent or not.
say_with_time	Outputs text along with how long it took to run its block. If the block returns an integer it assumes it is the number of rows affected.

For example, this migration:

```
class CreateProducts < ActiveRecord::Migration
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end

    say "Created a table"

    suppress_messages {add_index :products, :name}
    say "and an index!", true

    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

generates the following output

```
==  CreateProducts: migrating
```

```
==========================================
-- Created a table
   -> and an index!
-- Waiting for a while
   -> 10.0013s
   -> 250 rows
==  CreateProducts: migrated (10.0054s)
======================================
```

If you want Active Record to not output anything, then running `rake db:migrate VERBOSE=false` will suppress all output.

5 Changing Existing Migrations

Occasionally you will make a mistake when writing a migration. If you have already run the migration then you cannot just edit the migration and run the migration again: Rails thinks it has already run the migration and so will do nothing when you run `rake db:migrate`. You must rollback the migration (for example with `rake db:rollback`), edit your migration and then run `rake db:migrate` to run the corrected version.

In general, editing existing migrations is not a good idea. You will be creating extra work for yourself and your co-workers and cause major headaches if the existing version of the migration has already been run on production machines. Instead, you should write a new migration that performs the changes you require. Editing a freshly generated migration that has not yet been committed to source control (or, more generally, which has not been propagated beyond your development machine) is relatively harmless.

The `revert` method can be helpful when writing a new migration to undo previous migrations in whole or in part (see **Reverting Previous Migrations** above).

6 Using Models in Your Migrations

When creating or updating data in a migration it is often tempting to use one of your models. After all, they exist to provide easy access to the underlying data. This can be done, but some caution should be observed.

For example, problems occur when the model uses database columns which are (1) not currently in the database and (2) will be created by this or a subsequent migration.

Consider this example, where Alice and Bob are working on the same code base which contains a `Product` model:

Bob goes on vacation.

Alice creates a migration for the `products` table which adds a new column and initializes it:

```ruby
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  def change
    add_column :products, :flag, :boolean
    reversible do |dir|
      dir.up { Product.update_all flag: false }
```

```
    end
  end
end
```

She also adds a validation to the `Product` model for the new column:

```
# app/models/product.rb

class Product < ActiveRecord::Base
  validates :flag, inclusion: { in: [true, false] }
end
```

Alice adds a second migration which adds another column to the `products` table and initializes it:

```
# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  def change
    add_column :products, :fuzz, :string
    reversible do |dir|
      dir.up { Product.update_all fuzz: 'fuzzy' }
    end
  end
end
```

She also adds a validation to the `Product` model for the new column:

```
# app/models/product.rb

class Product < ActiveRecord::Base
  validates :flag, inclusion: { in: [true, false] }
  validates :fuzz, presence: true
end
```

Both migrations work for Alice.

Bob comes back from vacation and:

- Updates the source - which contains both migrations and the latest version of the Product model.
- Runs outstanding migrations with `rake db:migrate`, which includes the one that updates the `Product` model.

The migration crashes because when the model attempts to save, it tries to validate the second added column, which is not in the database when the *first* migration runs:

```
rake aborted!
An error has occurred, this and all later migrations canceled:
```

```
undefined method `fuzz' for #<Product:0x000001049b14a0>
```

A fix for this is to create a local model within the migration. This keeps Rails from running the validations, so that the migrations run to completion.

When using a local model, it's a good idea to call `Product.reset_column_information` to refresh the Active Record cache for the `Product` model prior to updating data in the database.

If Alice had done this instead, there would have been no problem:

```ruby
# db/migrate/20100513121110_add_flag_to_product.rb

class AddFlagToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
  end

  def change
    add_column :products, :flag, :boolean
    Product.reset_column_information
    reversible do |dir|
      dir.up { Product.update_all flag: false }
    end
  end
end
```

```ruby
# db/migrate/20100515121110_add_fuzz_to_product.rb

class AddFuzzToProduct < ActiveRecord::Migration
  class Product < ActiveRecord::Base
  end

  def change
    add_column :products, :fuzz, :string
    Product.reset_column_information
    reversible do |dir|
      dir.up { Product.update_all fuzz: 'fuzzy' }
    end
  end
end
```

There are other ways in which the above example could have gone badly.

For example, imagine that Alice creates a migration that selectively updates the `description` field on certain products. She runs the migration, commits the code, and then begins working on the next feature, which is to add a new column `fuzz` to the products table.

She creates two migrations for this new feature, one which adds the new column, and a second which selectively updates the `fuzz` column based on other product attributes.

These migrations run just fine, but when Bob comes back from his vacation and calls `rake db:migrate` to run all the outstanding migrations, he gets a subtle bug: The descriptions have defaults, and the `fuzz` column is present, but `fuzz` is `nil` on all products.

The solution is again to use `Product.reset_column_information` before referencing the Product model in a migration, ensuring the Active Record's knowledge of the table structure is current before manipulating data in those records.

7 Schema Dumping and You

7.1 What are Schema Files for?

Migrations, mighty as they may be, are not the authoritative source for your database schema. That role falls to either `db/schema.rb` or an SQL file which Active Record generates by examining the database. They are not designed to be edited, they just represent the current state of the database.

There is no need (and it is error prone) to deploy a new instance of an app by replaying the entire migration history. It is much simpler and faster to just load into the database a description of the current schema.

For example, this is how the test database is created: the current development database is dumped (either to `db/schema.rb` or `db/structure.sql`) and then loaded into the test database.

Schema files are also useful if you want a quick look at what attributes an Active Record object has. This information is not in the model's code and is frequently spread across several migrations, but the information is nicely summed up in the schema file. The **annotate_models** gem automatically adds and updates comments at the top of each model summarizing the schema if you desire that functionality.

7.2 Types of Schema Dumps

There are two ways to dump the schema. This is set in `config/application.rb` by the `config.active_record.schema_format` setting, which may be either `:sql` or `:ruby`.

If `:ruby` is selected then the schema is stored in `db/schema.rb`. If you look at this file you'll find that it looks an awful lot like one very big migration:

```ruby
ActiveRecord::Schema.define(version: 20080906171750) do
  create_table "authors", force: true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", force: true do |t|
    t.string   "name"
    t.text "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string "part_number"
  end
end
```

In many ways this is exactly what it is. This file is created by inspecting the database and expressing its structure using `create_table`, `add_index`, and so on. Because this is database-independent, it could be loaded into any database that Active Record supports. This could be very useful if you were to distribute an application that is able to run against multiple databases.

There is however a trade-off: `db/schema.rb` cannot express database specific items such as foreign key constraints, triggers, or stored procedures. While in a migration you can execute custom SQL statements, the schema dumper cannot reconstitute those statements from the database. If you are using features like this, then you should set the schema format to `:sql`.

Instead of using Active Record's schema dumper, the database's structure will be dumped using a tool specific to the database (via the `db:structure:dump` Rake task) into `db/structure.sql`. For example, for PostgreSQL, the `pg_dump` utility is used. For MySQL, this file will contain the output of `SHOW CREATE TABLE` for the various tables.

Loading these schemas is simply a question of executing the SQL statements they contain. By definition, this will create a perfect copy of the database's structure. Using the `:sql` schema format will, however, prevent loading the schema into a RDBMS other than the one used to create it.

7.3 Schema Dumps and Source Control

Because schema dumps are the authoritative source for your database schema, it is strongly recommended that you check them into source control.

8 Active Record and Referential Integrity

The Active Record way claims that intelligence belongs in your models, not in the database. As such, features such as triggers or foreign key constraints, which push some of that intelligence back into the database, are not heavily used.

Validations such as `validates :foreign_key, uniqueness: true` are one way in which models can enforce data integrity. The `:dependent` option on associations allows models to automatically destroy child objects when the parent is destroyed. Like anything which operates at the application level, these cannot guarantee referential integrity and so some people augment them with foreign key constraints in the database.

Although Active Record does not provide any tools for working directly with such features, the `execute` method can be used to execute arbitrary SQL. You can also use a gem like **foreigner** which adds foreign key support to Active Record (including support for dumping foreign keys in `db/schema.rb`).

9 Migrations and Seed Data

Some people use migrations to add data to the database:

```
class AddInitialProducts < ActiveRecord::Migration
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A
product.")
    end
  end
```

```
  def down
    Product.delete_all
  end
end
```

However, Rails has a 'seeds' feature that should be used for seeding a database with initial data. It's a really simple feature: just fill up `db/seeds.rb` with some Ruby code, and run `rake db:seed`:

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A product.")
end
```

This is generally a much cleaner way to set up the database of a blank application.

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our **documentation contributions** section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check **Edge Guides** first to verify if the issues are already fixed or not on the master branch. Check the **Ruby on Rails Guides Guidelines** for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the **rubyonrails-docs mailing list**.