**DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.**

# 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 a 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 rails commands that manipulate migrations and your schema.
- How migrations relate to `schema.rb`.

---

## Migration Overview

Migrations are a convenient way to alter your database schema over time in a consistent 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[7.1]
  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.

NOTE: 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[7.1]
  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[7.1]
  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
```

## Creating a Migration

### 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 appropriately named empty migration:

```
class AddPartNumberToProducts < ActiveRecord::Migration[7.1]
  def change
  end
end
```

This generator can do much more than append a timestamp to the file name. Based on naming conventions and additional (optional) arguments it can also start fleshing out the migration.

If the migration name is of the form "AddColumnToTable" or "RemoveColumnFromTable" 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[7.1]
  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 the appropriate `add_column` and `add_index` statements:

```
class AddPartNumberToProducts < ActiveRecord::Migration[7.1]
  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[7.1]
  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[7.1]
  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[7.1]
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number

      t.timestamps
    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 example,

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

generates the following `add_reference` call:

```
class AddUserRefToProducts < ActiveRecord::Migration[7.1]
  def change
    add_reference :products, :user, foreign_key: true
  end
end
```

This migration will create a `user_id` column. References are a shorthand for creating columns, indexes, foreign keys, or even polymorphic association columns.

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

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

will produce the following migration:

```
class CreateJoinTableCustomerProduct < ActiveRecord::Migration[7.1]
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

### Model Generators

The model, resource, 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[7.1]
  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.

### Passing Modifiers

Some commonly used type modifiers can be passed directly on the command line. They are enclosed by curly braces and follow the field type:

For instance, running:

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

will produce a migration that looks like this

```ruby
class AddDetailsToProducts < ActiveRecord::Migration[7.1]
  def change
    add_column :products, :price, :decimal, precision: 5, scale: 2
    add_reference :products, :supplier, polymorphic: true
  end
end
```

TIP: Have a look at the generators help output for further details.

## Writing a Migration

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

### 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, resource, or scaffold generator. A typical use would be

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

which creates a `products` table with a column called `name`.

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:

```ruby
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.

An index can be created on the columns created within the `create_table` block by passing true or an options hash to the `:index` option:

```ruby
create_table :users do |t|
  t.string :name, index: true
  t.string :email, index: { unique: true, name: 'unique_emails' }
end
```

Also you can pass the `:comment` option with any description for the table that will be stored in database itself and can be viewed with database administration tools, such as MySQL Workbench or PgAdmin III. It's highly recommended to specify comments in migrations for applications with large databases as it helps people to understand data model and generate documentation. Currently only the MySQL and PostgreSQL adapters support comments.

### Creating a Join Table

The migration method `create_join_table` creates an HABTM (has and belongs to many) 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 }
```

By default, the name of the join table comes from the union of the first two arguments provided to create_join_table, in alphabetical order. To customize the name of the table, provide a `:table_name` option:

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

creates 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
```

### 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.

### Changing Columns

Like the `remove_column` and `add_column` Rails provides the `change_column` migration method.

```
change_column :products, :part_number, :text
```

This changes the column `part_number` on products table to be a `:text` field. Note that `change_column` command is irreversible.

Besides `change_column`, the `change_column_null` and `change_column_default` methods are used specifically to change a not null constraint and default values of a column.

```
change_column_null :products, :name, false
change_column_default :products, :approved, from: true, to: false
```

This sets `:name` field on products to a `NOT NULL` column and the default value of the `:approved` field from true to false.

NOTE: You could also write the above `change_column_default` migration as `change_column_default :products, :approved, false`, but unlike the previous example, this would make your migration irreversible.

### Column Modifiers

Column modifiers can be applied when creating or changing a column:

- `comment` Adds a comment for the column.
- `collation` Specifies the collation for a `string` or `text` column.
- `default` Allows to set a default value on the column. Note that if you are using a dynamic value (such as a date), the default will only be calculated the first time (i.e. on the date the migration is applied). Use `nil` for `NULL`.
- `limit` Sets the maximum number of characters for a `string` column and the maximum number of bytes for `text/binary/integer` columns.
- `null` Allows or disallows `NULL` values in the column.
- `precision` Specifies the precision for `decimal/numeric/datetime/time` columns.
- `scale` Specifies the scale for the `decimal` and `numeric` columns, representing the number of digits after the decimal point.

NOTE: For `add_column` or `change_column` there is no option for adding indexes. They need to be added separately using `add_index`.

Some adapters may support additional options; see the adapter specific API docs for further information.

NOTE: `null` and `default` cannot be specified via command line.

### References

The `add_reference` method allows the creation of an appropriately named column.

```
add_reference :users, :role
```

This migration will create a `role_id` column in the users table. It creates an index for this column as well, unless explicitly told not with the `index: false` option:

```
add_reference :users, :role, index: false
```

The method `add_belongs_to` is an alias of `add_reference`.

```
add_belongs_to :taggings, :taggable, polymorphic: true
```

The polymorphic option will create two columns on the taggings table which can be used for polymorphic associations: `taggable_type` and `taggable_id`.

A foreign key can be created with the `foreign_key` option.

```
add_reference :users, :role, foreign_key: true
```

For more `add_reference` options, visit the API documentation.

References can also be removed:

```
remove_reference :products, :user, foreign_key: true, index: false
```

### Foreign Keys

While it's not required you might want to add foreign key constraints to guarantee referential integrity.

```
add_foreign_key :articles, :authors
```

This `add_foreign_key` call adds a new constraint to the `articles` table. The constraint guarantees that a row in the `authors` table exists where the `id` column matches the `articles.author_id`.

If the `from_table` column name cannot be derived from the `to_table` name, you can use the `:column` option. Use the `:primary_key` option if the referenced primary key is not `:id`.

For example, to add a foreign key on `articles.reviewer` referencing `authors.email`:

```
add_foreign_key :articles, :authors, column: :reviewer, primary_key: :email
```

`add_foreign_key` also supports options such as `name`, `on_delete`, `if_not_exists`, `validate`, and `deferrable`.

NOTE: Active Record only supports single column foreign keys. `execute` and `structure.sql` are required to use composite foreign keys. See Schema Dumping and You.

Foreign keys can also be removed:

```
# let Active Record figure out the column name
remove_foreign_key :accounts, :branches

# remove foreign key for a specific column
remove_foreign_key :accounts, column: :owner_id
```

### 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=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`).

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

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

`remove_column` is reversible if you supply the column type as the third argument. Provide the original column options too, otherwise Rails can't recreate the column exactly when rolling back:

```
remove_column :posts, :slug, :string, null: false, default: ''
```

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.

### 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 and what else to do when reverting it. For example:

```
class ExampleMigration < ActiveRecord::Migration[7.1]
  def change
    create_table :distributors do |t|
      t.string :zipcode
    end

    reversible do |dir|
      dir.up do
        # add a CHECK constraint
        execute <<-SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO INHERIT;
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        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 `distributors` 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.

### 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 perform 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[7.1]
  def up
```

```
    create_table :distributors do |t|
      t.string :zipcode
    end

    # add a CHECK constraint
    execute <<-SQL
      ALTER TABLE distributors
        ADD CONSTRAINT zipchk
        CHECK (char_length(zipcode) = 5);
    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 distributors
        DROP CONSTRAINT zipchk
    SQL

    drop_table :distributors
  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.

### Reverting Previous Migrations

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

```
require_relative "20121212123456_example_migration"

class FixupExampleMigration < ActiveRecord::Migration[7.1]
  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 use Active Record validations, in place of the `CHECK` constraint, to verify the zipcode.

```
class DontUseConstraintForZipcodeValidationMigration < ActiveRecord::Migration[7.1]
  def change
    revert do
      # copy-pasted code from ExampleMigration
      reversible do |dir|
        dir.up do
          # add a CHECK constraint
          execute <<-SQL
            ALTER TABLE distributors
              ADD CONSTRAINT zipchk
                CHECK (char_length(zipcode) = 5);
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE distributors
              DROP CONSTRAINT zipchk
          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`.

## Running Migrations

Rails provides a set of rails commands to run certain sets of migrations.

The very first migration related rails command you will use will probably be `bin/rails 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` command also invokes the `db:schema:dump` command, 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/rails 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.

### 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/rails 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/rails db:rollback STEP=3
```

will revert the last 3 migrations.

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

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

Neither of these rails commands do anything you could not do with `db:migrate`. They are there for convenience, since you do not need to explicitly specify the version to migrate to.

### Setup the Database

The `bin/rails db:setup` command will create the database, load the schema, and initialize it with the seed data.

### Resetting the Database

The `bin/rails db:reset` command will drop the database and set it up again. This is functionally equivalent to `bin/rails db:drop db:setup`.

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

### Running Specific Migrations

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

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

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

already been run.

### Running Migrations in Different Environments

By default running `bin/rails 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/rails db:migrate RAILS_ENV=test
```

### 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[7.1]
  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 `bin/rails db:migrate VERBOSE=false` will suppress all output.

## 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 `bin/rails db:migrate`. You must rollback the migration (for example with `bin/rails db:rollback`), edit your migration, and then run `bin/rails 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).

## Schema Dumping and You

### What are Schema Files for?

Migrations, mighty as they may be, are not the authoritative source for your database schema. Your database remains the authoritative source. By default, Rails generates `db/schema.rb` which attempts to capture the current state of your database schema.

It tends to be faster and less error prone to create a new instance of your application's database by loading the schema file via `bin/rails db:schema:load` than it is to replay the entire migration history. Old migrations may fail to apply correctly if those migrations use changing external dependencies or rely on application code which evolves separately from your migrations.

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.

### Types of Schema Dumps

The format of the schema dump generated by Rails is controlled by the `config.active_record.schema_format` setting in `config/application.rb`. By default, the format is `:ruby`, but can also be set to `:sql`.

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[7.1].define(version: 2008_09_06_171750) 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.

`db/schema.rb` cannot express everything your database may support such as triggers, sequences, stored procedures, etc. While migrations may use `execute` to create database constructs that are not supported by the Ruby migration DSL, these constructs may not be able to be reconstituted by the schema dumper. If you are using features like these, you should set the schema format to `:sql` in order to get an accurate schema file that is useful to create new database instances.

When the schema format is set to `:sql`, the database structure will be dumped using a tool specific to the database into `db/structure.sql`. For example, for PostgreSQL, the `pg_dump` utility is used. For MySQL and MariaDB, this file will contain the output of `SHOW CREATE TABLE` for the various tables.

To load the schema from `db/structure.sql`, run `bin/rails db:schema:load`. Loading this file is done by executing the SQL statements it contains. By definition, this will create a perfect copy of the database's structure.

### Schema Dumps and Source Control

Because schema files are commonly used to create new databases, it is strongly recommended that you check your schema file into source control.

Merge conflicts can occur in your schema file when two branches modify schema. To resolve these conflicts run `bin/rails db:migrate` to regenerate the schema file.

## 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 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 all the tools for working directly with such features, the `execute` method can be used to execute arbitrary SQL.

## Migrations and Seed Data

The main purpose of Rails' migration feature is to issue commands that modify the schema using a consistent process. Migrations can also be used to add or modify data. This is useful in an existing database that can't be destroyed and recreated, such as a production database.

```
class AddInitialProducts < ActiveRecord::Migration[7.1]
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description: "A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

To add initial data after a database is created, Rails has a built-in 'seeds' feature that speeds up the process. This is especially useful when reloading the database frequently in development and test environments. To get started with this feature, fill up `db/seeds.rb` with some Ruby code, and run `bin/rails 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.

## Old Migrations

The `db/schema.rb` or `db/structure.sql` is a snapshot of the current state of your database and is the authoritative source for rebuilding that database. This makes it possible to delete old migration files.

When you delete migration files in the `db/migrate/` directory, any environment where `bin/rails db:migrate` was run when those files still existed will hold a reference to the migration timestamp specific to them inside an internal Rails database table named `schema_migrations`. This table is used to keep track of whether migrations have been executed in a specific environment.

If you run the `bin/rails db:migrate:status` command, which displays the status (up or down) of each migration, you should see `********** NO FILE **********` displayed next to any deleted migration file which was once executed on a specific environment but can no longer be found in the `db/migrate/` directory.

There's a caveat, though. Rake tasks to install migrations from engines are idempotent. Migrations present in the parent application due to a previous installation are skipped, and missing ones are copied with a new leading timestamp. If you deleted old engine migrations and ran the install task again, you'd get new files with new timestamps, and `db:migrate` would attempt to run them again.

Thus, you generally want to preserve migrations coming from engines. They have a special comment like this:

```
# This migration comes from blorgh (originally 20210621082949)
```