In this chapter, you'll see:

- Naming migration files
- Renaming and columns
- Creating and renaming tables
- Defining indices and keys
- Using native SQL

CHAPTER 23

Migrations

Rails encourages an agile, iterative style of development. We don't expect to get everything right the first time. Instead, we write tests and interact with our customers to refine our understanding as we go.

For that to work, we need a supporting set of practices. We write tests to help us design our interfaces and to act as a safety net when we change things, and we use version control to store our application's source files, allowing us to undo mistakes and to monitor what changes day to day.

But there's another area of the application that changes, an area that we can't directly manage using version control. The database schema in a Rails application constantly evolves as we progress through the development: we add a table here, rename a column there, and so on. The database changes in step with the application's code.

With Rails, each of those steps is made possible through the use of a *migration*. You saw this in use throughout the development of the Depot application, starting when we created the first `products` table in Generating the Scaffold, on page 70, and when we performed such tasks as adding a quantity to the `line_items` table in Iteration E1: Creating a Smarter Cart, on page 127. Now it's time to dig deeper into how migrations work and what else you can do with them.

Creating and Running Migrations

A migration is simply a Ruby source file in your application's `db/migrate` directory. Each migration file's name starts with a number of digits (typically fourteen) and an underscore. Those digits are the key to migrations, because they define the sequence in which the migrations are applied—they're the individual migration's version number.

The version number is the Coordinated Universal Time (UTC) timestamp at the time the migration was created. These numbers contain the four-digit year, followed by two digits each for the month, day, hour, minute, and second, all based on the mean solar time at the Royal Observatory in Greenwich, London. Because migrations tend to be created relatively infrequently and the accuracy is recorded down to the second, the chances of any two people getting the same timestamp is vanishingly small. And the benefit of having timestamps that can be deterministically ordered far outweighs the miniscule risk of this occurring.

Here's what the db/migrate directory of our Depot application looks like:

```
depot> ls db/migrate
20221207000001_create_products.rb
20221207000002_create_carts.rb
20221207000003_create_line_items.rb
20221207000004_add_quantity_to_line_items.rb
20221207000005_combine_items_in_cart.rb
20221207000006_create_orders.rb
20221207000007_add_order_id_to_line_item.rb
20221207000008_create_users.rb
```

Although you could create these migration files by hand, it's easier (and less error prone) to use a generator. As we saw when we created the Depot application, two generators create migration files:

- The *model* generator creates a migration to in turn create the table associated with the model (unless you specify the --skip-migration option). As the example that follows shows, creating a model called discount also creates a migration called *yyyyMMddhhmmss*_create_discounts.rb:

```
depot> bin/rails generate model discount
    invoke  active_record
➤   create    db/migrate/20221207133549_create_discounts.rb
    create    app/models/discount.rb
    invoke    test_unit
    create      test/models/discount_test.rb
    create      test/fixtures/discounts.yml
```

- You can also generate a migration on its own.

```
depot> bin/rails generate migration add_price_column
    invoke  active_record
➤   create    db/migrate/20221207133814_add_price_column.rb
```

Later, starting in Anatomy of a Migration, on page 398, we'll see what goes in the migration files. But for now, let's jump ahead a little in the workflow and see how to run migrations.

Running Migrations

Migrations are run using the db:migrate Rake task:

```
depot> bin/rails db:migrate
```

To see what happens next, let's dive down into the internals of Rails.

The migration code maintains a table called schema_migrations inside every Rails database. This table has just one column, called version, and it will have one row per successfully applied migration.

When you run bin/rails db:migrate, the task first looks for the schema_migrations table. If it doesn't yet exist, it'll be created.

The migration code then looks at the migration files in db/migrate and skips from consideration any that have a version number (the leading digits in the filename) that's already in the database. It then proceeds to apply the remainder of the migrations, creating a row in the schema_migrations table for each.

If we were to run migrations again at this point, nothing much would happen. Each of the version numbers of the migration files would match with a row in the database, so there'd be no migrations to apply.

But if we subsequently create a new migration file, it will have a version number not in the database. This is true even if the version number was *before* one or more of the already applied migrations. This can happen when multiple users are using a version control system to store the migration files. If we then run migrations, this new migration file—and only this migration file—will be executed. This may mean that migrations are run out of order, so you might want to take care and ensure that these migrations are independent. Or you might want to revert your database to a previous state and then apply the migrations in order.

You can force the database to a specific version by supplying the VERSION= parameter to the rake db:migrate command:

```
depot> bin/rails db:migrate VERSION=20221207000009
```

If the version you give is greater than any of the migrations that have yet to be applied, these migrations will be applied.

If, however, the version number on the command line is less than one or more versions listed in the schema_migrations table, something different happens. In these circumstances, Rails looks for the migration file whose number matches the database version and *undoes* it. It repeats this process until there are no more versions listed in the schema_migrations table that exceed the number you

specified on the command line. That is, the migrations are unapplied in reverse order to take the schema back to the version that you specify.

You can also redo one or more migrations:

```
depot> bin/rails db:migrate:redo STEP=3
```

By default, redo will roll back one migration and rerun it. To roll back multiple migrations, pass the STEP= parameter.

Anatomy of a Migration

Migrations are subclasses of the Rails class ActiveRecord::Migration. When necessary, migrations can contain up() and down() methods:

```
class SomeMeaningfulName < ActiveRecord::Migration
  def up
    # ...
  end

  def down
    # ...
  end
end
```

The name of the class, after all uppercase letters are downcased and preceded by an underscore, must match the portion of the filename after the version number. For example, the previous class could be found in a file named 20221207000017_some_meaningful_name.rb. No two migrations can contain classes with the same name.

The up() method is responsible for applying the schema changes for this migration, while the down() method undoes those changes. Let's make this more concrete. Here's a migration that adds an e_mail column to the orders table:

```
class AddEmailToOrders < ActiveRecord::Migration
  def up
    add_column :orders, :e_mail, :string
  end

  def down
    remove_column :orders, :e_mail
  end
end
```

See how the down() method undoes the effect of the up() method? You can also see a bit of duplication here. In many cases, Rails can detect how to automatically undo a given operation. For example, the opposite of add_column() is clearly remove_column(). In such cases, by simply renaming up() to change(), you can eliminate the need for a down():

```
class AddEmailToOrders < ActiveRecord::Migration
  def change
    add_column :orders, :e_mail, :string
  end
end
```

Now isn't that much cleaner?

Column Types

The third parameter to add_column specifies the type of the database column. In the prior example, we specified that the e_mail column has a type of :string. But what does this mean? Databases typically don't have column types of :string.

Remember that Rails tries to make your application independent of the underlying database; you could develop using SQLite 3 and deploy to Postgres if you wanted, for example. But different databases use different names for the types of columns. If you used a SQLite 3 column type in a migration, that migration might not work if applied to a Postgres database. So, Rails migrations insulate you from the underlying database type systems by using logical types. If we're migrating a SQLite 3 database, the :string type will create a column of type varchar(255). On Postgres, the same migration adds a column with the type char varying(255).

The types supported by migrations are :binary, :boolean, :date, :datetime, :decimal, :float, :integer, :string, :text, :time, and :timestamp. The default mappings of these types for the database adapters in Rails are shown in the following tables:

	db2	mysql	openbase	oracle
:binary	blob(32768)	blob	object	blob
:boolean	decimal(1)	tinyint(1)	boolean	number(1)
:date	date	date	date	date
:datetime	timestamp	datetime	datetime	date
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float	number
:integer	int	int(11)	integer	number(38)
:string	varchar(255)	varchar(255)	char(4096)	varchar2(255)
:text	clob(32768)	text	text	clob
:time	time	time	time	date
:timestamp	timestamp	datetime	timestamp	date

	postgresql	sqlite	sqlserver	sybase
:binary	bytea	blob	image	image
:boolean	boolean	boolean	bit	bit
:date	date	date	date	datetime
:datetime	timestamp	datetime	datetime	datetime
:decimal	decimal	decimal	decimal	decimal
:float	float	float	float(8)	float(8)
:integer	integer	integer	int	int
:string	(note 1)	varchar(255)	varchar(255)	varchar(255)
:text	text	text	text	text
:time	time	datetime	time	time
:timestamp	timestamp	datetime	datetime	timestamp

Using these tables, you could work out that a column declared to be :integer in a migration would have the underlying type integer in SQLite 3 and number(38) in Oracle.

You can use three options when defining most columns in a migration; decimal columns take an additional two options. Each of these options is given as a key: value pair. The common options are as follows:

null: true or false If false, the underlying column has a not null constraint added (if the database supports it). Note that this is independent of any presence: true validation, which may be performed at the model layer.

limit: size This sets a limit on the size of the field. It appends the string (*size*) to the database column type definition.

default: value This sets the default value for the column. Since it's performed by the database, you don't see this in a new model object when you initialize it or even when you save it. You have to reload the object from the database to see this value. Note that the default is calculated once, at the point the migration is run, so the following code will set the default column value to the date and time when the migration was run:

```
add_column :orders, :placed_at, :datetime, default: Time.now
```

In addition, decimal columns take the options :precision and :scale. The :precision option specifies the number of significant digits that will be stored, and the :scale option determines where the decimal point will be located in these digits (think of the scale as the number of digits after the decimal point). A decimal number with a precision of 5 and a scale of 0 can store numbers from -99,999

to +99,999. A decimal number with a precision of 5 and a scale of 2 can store the range -999.99 to +999.99.

The :precision and :scale parameters are optional for decimal columns. However, incompatibilities between different databases lead us to strongly recommend that you include the options for each decimal column.

Here are some column definitions using the migration types and options:

```
add_column :orders, :attn, :string, limit: 100
add_column :orders, :order_type,  :integer
add_column :orders, :ship_class, :string, null: false, default: 'priority'
add_column :orders, :amount, :decimal, precision: 8, scale: 2
```

Renaming Columns

When we refactor our code, we often change our variable names to make them more meaningful. Rails migrations allow us to do this to database column names too. For example, a week after we first added it, we might decide that e_mail isn't the best name for the new column. We can create a migration to rename it using the rename_column() method:

```
class RenameEmailColumn < ActiveRecord::Migration
  def change
    rename_column :orders, :e_mail, :customer_email
  end
end
```

As rename_column() is reversible, separate up() and down() methods aren't required in order to use it.

Note that the rename doesn't destroy any existing data associated with the column. Also be aware that renaming is not supported by all the adapters.

Changing Columns

change_column() Use the change_column() method to change the type of a column or to alter the options associated with a column. Use it the same way you'd use add_column, but specify the name of an existing column. Let's say that the order type column is currently an integer, but we need to change it to be a string. We want to keep the existing data, so an order type of 123 will become the string "123". Later, we'll use noninteger values such as "new" and "existing".

Changing from an integer column to a string is one line of code:

```
def up
  change_column :orders, :order_type, :string
end
```

However, the opposite transformation is problematic. We might be tempted to write the obvious down() migration:

```
def down
 change_column :orders, :order_type, :integer
end
```

But if our application has taken to storing data like "new" in this column, the down() method will lose it—"new" can't be converted to an integer. If that's acceptable, then the migration is acceptable as it stands. If, however, we want to create a one-way migration—one that can't be reversed—we'll want to stop the down migration from being applied. In this case, Rails provides a special exception that we can throw:

```
class ChangeOrderTypeToString < ActiveRecord::Migration
  def up
    change_column :orders, :order_type, :string, null: false
  end

  def down
    raise ActiveRecord::IrreversibleMigration
  end
end
```

ActiveRecord::IrreversibleMigration is also the name of the exception that Rails will raise if you attempt to call a method that can't be automatically reversed from within a change() method.

Managing Tables

So far we've been using migrations to manipulate the columns in existing tables. Now let's look at creating and dropping tables:

```
class CreateOrderHistories < ActiveRecord::Migration
  def change
    create_table :order_histories do |t|
      t.integer :order_id, null: false
      t.text :notes

      t.timestamps
    end
  end
end
```

create_table() takes the name of a table (remember, table names are plural) and a block. (It also takes some optional parameters that we'll look at in a minute.) The block is passed a table definition object, which we use to define the columns in the table.

Generally the call to drop_table() isn't needed, as create_table() is reversible. drop_table() accepts a single parameter, which is the name of the table to drop.

The calls to the various table definition methods should look familiar—they're similar to the add_column method we used previously, except these methods don't take the name of the table as the first parameter and the name of the method itself is the data type desired. This reduces repetition.

Note that we don't define the id column for our new table. Unless we say otherwise, Rails migrations automatically add a primary key called id to all tables they create. For a deeper discussion of this, see Primary Keys, on page 406.

The timestamps method creates both the created_at and updated_at columns, with the correct timestamp data type. Although there's no requirement to add these columns to any particular table, this is yet another example of Rails making it easy for a common convention to be implemented easily and consistently.

Options for Creating Tables

You can pass a hash of options as a second parameter to create_table. If you specify force: true, the migration will drop an existing table of the same name before creating the new one. This is a useful option if you want to create a migration that forces a database into a known state, but there's clearly a potential for data loss.

The temporary: true option creates a temporary table—one that goes away when the application disconnects from the database. This is clearly pointless in the context of a migration, but as we'll see later, it does have its uses elsewhere.

The options: "xxxx" parameter lets you specify options to your underlying database. They're added to the end of the CREATE TABLE statement, right after the closing parenthesis. Although this is rarely necessary with SQLite 3, it may at times be useful with other database servers. For example, some versions of MySQL allow you to specify the initial value of the autoincrementing id column. We can pass this in through a migration as follows:

```
create_table :tickets, options: "auto_increment = 10000" do |t|
  t.text :description
  t.timestamps
end
```

Behind the scenes, migrations will generate the following DDL from this table description when configured for MySQL:

```
CREATE TABLE "tickets" (
  "id" int(11) default null auto_increment primary key,
  "description" text,
  "created_at" datetime,
  "updated_at" datetime
) auto_increment = 10000;
```

Be careful when using the :options parameter with MySQL. The Rails MySQL database adapter sets a default option of ENGINE=InnoDB. This overrides any local defaults you have and forces migrations to use the InnoDB storage engine for new tables. Yet, if you override :options, you'll lose this setting; new tables will be created using whatever database engine is configured as the default for your site. You may want to add an explicit ENGINE=InnoDB to the options string to force the standard behavior in this case. You probably want to keep using InnoDB if you're using MySQL because this engine gives you transaction support. You might need this support in your application, and you'll definitely need it in your tests if you're using the default of transactional test fixtures.

Renaming Tables

If refactoring leads us to rename variables and columns, then it's probably not a surprise that we sometimes find ourselves renaming tables too. Migrations support the rename_table() method:

```
class RenameOrderHistories < ActiveRecord::Migration
  def change
    rename_table :order_histories, :order_notes
  end
end
```

Rolling back this migration undoes the change by renaming the table back.

Problems with rename_table

When we rename tables in migrations, a subtle problem arises.

For example, let's assume that in migration 4 we create the order_histories table and populate it with some data:

```
def up
  create_table :order_histories do |t|
   t.integer :order_id, null: false
   t.text :notes

   t.timestamps
  end

  order = Order.find :first
  OrderHistory.create(order_id: order, notes: "test")
end
```

Later, in migration 7, we rename the table order_histories to order_notes. At this point we'll also have renamed the model OrderHistory to OrderNote.

Now we decide to drop our development database and reapply all migrations. When we do so, the migrations throw an exception in migration 4: our application no longer contains a class called OrderHistory, so the migration fails.

One solution, proposed by Tim Lucas, is to create local dummy versions of the model classes needed by a migration within the migration. For example, the following version of the fourth migration will work even if the application no longer has an OrderHistory class:

```
class CreateOrderHistories < ActiveRecord::Migration
➤  class Order < ApplicationRecord::Base; end
➤  class OrderHistory < ApplicationRecord::Base; end

  def change
    create_table :order_histories do |t|
      t.integer :order_id, null: false
      t.text :notes

      t.timestamps
    end

   order = Order.find :first
   OrderHistory.create(order: order_id, notes: "test")
  end
end
```

This works as long as our model classes don't contain any additional functionality that would have been used in the migration—all we're creating here is a bare-bones version.

Defining Indices

Migrations can (and probably should) define indices for tables. For example, we might notice that once our application has a large number of orders in the database, searching based on the customer's name takes longer than we'd like. It's time to add an index using the appropriately named add_index() method:

```
class AddCustomerNameIndexToOrders < ActiveRecord::Migration
  def change
    add_index :orders, :name
  end
end
```

If we give add_index the optional parameter unique: true, a unique index will be created, forcing values in the indexed column to be unique.

By default the index will be given the name *index_table_on_column*. We can override this using the name: "somename" option. If we use the :name option when adding an index, we'll also need to specify it when removing the index.

We can create a *composite index*—an index on multiple columns—by passing an array of column names to add_index.

Indices are removed using the remove_index() method.

Primary Keys

Rails assumes every table has a numeric primary key (normally called id) and ensures the value of this column is unique for each new row added to a table. We'll rephrase that.

Rails doesn't work too well unless each table has a primary key that Rails can manage. By default, Rails will create numeric primary keys, but you can also use other types such as UUIDs, depending on what your actual database provides. Rails is less fussy about the name of the column. So for your average Rails application, our strong advice is to go with the flow and let Rails have its id column.

If you decide to be adventurous, you can start by using a different name for the primary key column (but keeping it as an incrementing integer). Do this by specifying a :primary_key option on the create_table call:

```
create_table :tickets, primary_key: :number do |t|
  t.text :description

  t.timestamps
end
```

This adds the number column to the table and sets it up as the primary key:

```
$ sqlite3 db/development.sqlite3 ".schema tickets"
CREATE TABLE tickets ("number" INTEGER PRIMARY KEY AUTOINCREMENT
NOT NULL, "description" text DEFAULT NULL, "created_at" datetime
DEFAULT NULL, "updated_at" datetime DEFAULT NULL);
```

The next step in the adventure might be to create a primary key that isn't an integer. Here's a clue that the Rails developers don't think this is a good idea: migrations don't let you do this (at least not directly).

Tables with No Primary Key

Sometimes we may need to define a table that has no primary key. The most common case in Rails is for *join tables*—tables with just two columns where each column is a foreign key to another table. To create a join table using migrations, we have to tell Rails not to automatically add an id column:

```
create_table :authors_books, id: false do |t|
  t.integer :author_id, null: false
  t.integer :book_id,   null: false
end
```

In this case, you might want to investigate creating one or more indices on this table to speed navigation between books and authors.

Advanced Migrations

Most Rails developers use the basic facilities of migrations to create and maintain their database schemas. But every now and then it's useful to push migrations just a bit further. This section covers some more advanced migration usage.

Using Native SQL

Migrations give you a database-independent way of maintaining your application's schema. However, if migrations don't contain the methods you need to be able to do what you need to do, you'll need to drop down to database-specific code. Rails provides two ways to do this. One is with options arguments to methods like add_column(). The second is the execute() method.

When you use options or execute(), you might well be tying your migration to a specific database engine, because any SQL you provide in these two locations uses your database's native syntax.

An example of where you might need to use raw SQL is if you're creating a custom data type inside your database. Postgres, for example, allows you to specify *enumerated types*. Enumerated types work just fine with Rails; but to create them in a migration, you have to use SQL and thus execute(). Suppose we wanted to create an enumerated type for the various pay types we supported in our checkout form (which we created in Chapter 12, Task G: Check Out!, on page 165):

```
class AddPayTypes < ActiveRecord::Migrations[6.0]
  def up
    execute %{
      CREATE TYPE
        pay_type
      AS ENUM (
        'check',
        'credit card',
        'purchase order'
      )
    }
  end
```

```
  def down
    execute "DROP TYPE pay_type"
  end
end
```

Note that if you need to model your database using execute(), you should consider changing your schema dump format from "ruby" to "SQL," as outlined in the Rails Guide.[1] The schema dump is used during tests to create an empty database with the same schema you're using in production.

Custom Messages and Benchmarks

Although not exactly an advanced migration, something that's useful to do within advanced migrations is to output our own messages and benchmarks. We can do this with the say_with_time() method:

```
def up
  say_with_time "Updating prices..." do
    Person.all.each do |p|
      p.update_attribute :price, p.lookup_master_price
    end
  end
 end
```

say_with_time() prints the string passed before the block is executed and prints the benchmark after the block completes.

When Migrations Go Bad

Migrations suffer from one serious problem. The underlying DDL statements that update the database schema are not transactional. This isn't a failing in Rails—most databases don't support the rolling back of create table, alter table, and other DDL statements.

Let's look at a migration that tries to add two tables to a database:

```
class ExampleMigration < ActiveRecord::Migration
  def change
    create_table :one do ...
    end
    create_table :two do ...
    end
  end
end
```

1. http://guides.rubyonrails.org/active_record_migrations.html#schema-dumping-and-you

In the normal course of events, the up() method adds tables, one and two, and the down() method removes them.

But what happens if there's a problem creating the second table? We'll end up with a database containing table one but not table two. We can fix whatever the problem is in the migration, but now we can't apply it—if we try, it will fail because table one already exists.

We could try to roll the migration back, but that won't work. Because the original migration failed, the schema version in the database wasn't updated, so Rails won't try to roll it back.

At this point, you could mess around and manually change the schema information and drop table one. But it probably isn't worth it. Our recommendation in these circumstances is simply to drop the entire database, re-create it, and apply migrations to bring it back up-to-date. You'll have lost nothing, and you'll know you have a consistent schema.

All this discussion suggests that migrations are dangerous to use on production databases. Should you run them? We really can't say. If you have database administrators in your organization, it'll be their call. If it's up to you, you'll have to weigh the risks. But if you decide to go for it, you really must back up your database first. Then you can apply the migrations by going to your application's directory on the machine with the database role on your production servers and executing this command:

```
depot> RAILS_ENV=production bin/rails db:migrate
```

This is one of those times where the legal notice at the start of this book kicks in. We're not liable if this deletes your data.

Schema Manipulation Outside Migrations

All the migration methods described so far in this chapter are also available as methods on Active Record connection objects and so are accessible within the models, views, and controllers of a Rails application.

For example, you might have discovered that a particular long-running report runs a lot faster if the orders table has an index on the city column. But that index isn't needed during the day-to-day running of the application, and tests have shown that maintaining it slows the application appreciably.

Let's write a method that creates the index, runs a block of code, and then drops the index. This could be a private method in the model or could be implemented in a library:

```
def run_with_index(*columns)
  connection.add_index(:orders, *columns)
  begin
    yield
  ensure
    connection.remove_index(:orders, *columns)
  end
end
```

The statistics-gathering method in the model can use this as follows:

```
def get_city_statistics
  run_with_index(:city) do
    # .. calculate stats
  end
end
```

What We Just Did

While we had been informally using migrations throughout the development of the Depot application and even into deployment, in this chapter we saw how migrations are the basis for a principled and disciplined approach to configuration management of the schema for your database.

You learned how to create, rename, and delete columns and tables, to manage indices and keys, to apply and back out entire sets of changes, and even to add your own custom SQL into the mix, all in a completely reproducible manner.

At this point we've covered the externals of Rails. The next chapter is going to show a few more involved ways of customizing Rails to demonstrate just how flexible Rails can be when you need it. We'll see how to use RSpec for testing, use Slim instead of ERB for templating, and use Webpack to manage your CSS.