Ruby Magic

Dissecting Rails Migrations

Prathamesh Sonpatki on Apr 14, 2020

In today's post, we'll take a deep dive into Rails migrations. We'll break down the migration into different pieces, and in the process, learn how to write an effective migration. We'll learn how to write migrations for multiple databases, as well as how to handle failed migrations and techniques of performing rollbacks.

To understand the whole post, you'll need to have a basic understanding of databases and Rails.

Migrations 101

Migrations in Rails allow us to evolve the database over the lifetime of an application. Migrations allow us to write plain Ruby code to alter the state of the database by providing an elegant DSL. We don't have to write database-specific SQL since migrations provide abstractions to manipulate the database and take care of nitty-gritty details of converting the DSL into database-specific SQL queries behind the scene. Migrations also get out of our way and provide ways of executing raw SQL on the database, if such need arises.

Twenty Thousand Leagues Into a Rails Database Migration

We can create tables, add or remove columns and add indexes on columns using the migrations.

Every Rails app has a special directory— `db/migrate` —where all migrations are stored.

Let’s start with a migration that creates the table `events` into our database.

```
$ rails g migration CreateEvents category:string
```

This command generates a timestamped file `20200405103635_create_events.rb` in the `db/migrate` directory. The contents of the file are as follows.

```
class CreateEvents < ActiveRecord::Migration[6.0]
  def change
    create_table :events do |t|
      t.string :category

      t.timestamps
    end
  end
end
```

Let’s break down this migration file.

- Every migration file that Rails generates has a timestamp that is present in the filename. This timestamp is important and is used by Rails to confirm whether a migration has run or not, as we’ll see later.
- The migration contains a class that inherits from `ActiveRecord::Migration[6.0]` . As I’m using Rails 6, the migration superclass has `[6.0]` . If I was using Rails 5.2, then the superclass would be `ActiveRecord::Migration[5.2]` . Later, we’ll discuss why the Rails version is part of the superclass name.
- The migration has a method `change` which contains the DSL code that manipulates the database. In this case, the `change` method is creating an `events` table with a column `category` of type `string` .
- The migration uses the code `t.timestamps` to add timestamps `created_at` and `updated_at` to the `events` table.

When this migration is run using the `rails db:migrate` command, it will create an `events` table with a `category` column of type `string` and timestamp columns `created_at` and `updated_at` .

> The actual database column type will be varchar or text, depending on the database.

IMPORTANCE OF MIGRATION TIMESTAMPS AND THE SCHEMA_MIGRATION TABLE

Every time a migration is generated using the `rails g migration` command, Rails generates the migration file with a unique timestamp. The timestamp is in the format `YYYYMMDDHHMMSS` . Whenever a migration is run, Rails inserts the migration timestamp into an internal table `schema_migrations` . This table is created by Rails when we run our first migration. The table only has the column `version` , which is also its primary key. This is the structure of the `schema_migrations` table.

```
CREATE TABLE IF NOT EXISTS "schema_migrations" ("version" varchar NOT NULL PRIMARY KEY);
```

Now that we have run the migration for creating the `events` table, let's see if Rails has stored a timestamp of this migration in the `schema_migrations` table.

```
sqlite> select * from schema_migrations;
20200405103635
```

If we run the migrations again, Rails will first check if an entry exists in the `schema_migrations` table with the timestamp of the migration file, and only execute it if there is no such entry. This ensures that we can incrementally add changes to the database over time and a migration will run only once on the database.

DATABASE SCHEMA

As we run more and more migrations, the database schema keeps evolving. Rails stores the most recent database schema in the file `db/schema.rb` . This file is the Ruby representation of all the migrations run on your database over the life of the application. Because of this file, we don't need to keep old migrations files in the codebase. Rails provides tasks to `dump` the latest schema from the database into `schema.rb` and `load` the schema into a database from the `schema.rb`. So older migrations can be safely deleted from the codebase. The loading of the schema into the database is also faster compared to running each and every migration every time we set up the application.

> Rails also provides a way to store database schema in SQL format. We already have an article to compare the two formats. You can read more about it here.

RAILS VERSION IN THE MIGRATION

Every migration that we generate has the Rails version as part of the superclass. So a migration generated by a Rails 6 app has the superclass `ActiveRecord::Migration[6.0]` whereas a migration generated by Rails 5.2 app has the superclass `ActiveRecord::Migration[5.2]`. If you have an old app with Rails 4.2 or below, you'll notice that there is no version in the superclass. The superclass is just `ActiveRecord::Migration`.

The Rails version was added to the migration superclass in Rails 5. This basically ensures that the migration API can evolve over time without breaking migrations generated by older versions of Rails.

Let's look deeper into this by looking at the same migration for creating an `events` table in a Rails 4.2 app.

```
class CreateEvents < ActiveRecord::Migration
```

```
      t.timestamps null: false
    end
  end
end
```

If we look at the schema of the `events` table generated by a Rails 6 migration, we can see that the `NOT NULL` constraint for the timestamps columns exist.

```
sqlite> .schema events
```

```
CREATE TABLE IF NOT EXISTS "events" ("id" integer PRIMARY KEY AUTOINCREMENT NOT NULL, "cate
```

This is because, starting from Rails 5 onward, the migration API automatically adds a `NOT NULL` constraint to the timestamp columns without a need to add it explicitly in the migration file. The Rails version in the superclass name ensures that the migration uses the migration API of the Rails version for which the migration was generated. This allows Rails to maintain backward compatibility with the older migrations, at the same time evolving the migrations API.

CHANGING THE DATABASE SCHEMA

The `change` method is the primary method in a migration. When a migration gets run, it calls the `change` method and executes the code inside it.

Along with `create_table` , Rails also provides another powerful method— `change_table` . As the name suggests, it is used to alter the schema of an existing table.

```
def change
  change_table :events do |t|
    t.remove :category
    t.string :event_type
    t.boolean :active, default: false
  end
end
```

This migration will remove the `category` column from the `events` table, add a new string column `events_type` and a new boolean column `active` with the default value of `false` .

Rails also provides a lot of other helper methods which can be used inside a migration such as:

- `change_column`
- `add_index`
- `remove_index`
- `rename_table`

and many more. All the methods that can be used with change can be found here

TIMESTAMPS

We saw that `t.timestamps` was added to the migration by Rails and it added the columns `created_at` and `updated_at` to the `events` table. These special columns are used by Rails to keep track of when a record is created and updated. Rails adds values to these columns when a record is created and makes sure to update them when the record is updated. These columns help us in tracking the lifetime of a database record.

> The `updated_at` column is not updated when we execute the `updated_all` method from Rails.

HANDLING FAILURES

Migrations are not bulletproof. They can fail. The reason might be wrong syntax or an invalid database query. Whatever the reason, we have to handle the failure and recover from it so that the database doesn't go into an inconsistent state. Rails solves this problem by running each migration inside a transaction. If the migration fails, then the transaction is rolled back. This ensures that the database does not go into an inconsistent state.

> This is only done for databases that support transactions for updating database schema. They are known as Data Definition Language(DDL) transactions. MySQL and PostgreSQL both support DDL transactions.

Sometimes, we don't want to execute certain migrations inside a transaction. A simple example is when adding a concurrent index in PostgreSQL. Such migrations can't be executed inside a DDL transaction as PostgreSQL tries to add the index without acquiring locks on the table so that we can add the index on a live production database without taking the database down. Rails provides a way to opt-out of transactions inside a migration in the form of `disable_ddl_transactions!` .

```ruby
def change
  disable_ddl_transactions!

  add_index :events, :user_id, algorithm: :concurrently
```

This will not run the migration inside a transaction. If such a migration fails, we need to recover it ourselves. In this case, we can either `REINDEX` or remove the index and try to add it again.

REVERSIBLE MIGRATIONS

Rails allows us to rollback changes to the database with the following command.

```
rails db:rollback
```

This command reverts the last migration that was run on the database. If the migration added a column `event_type` then the rollback will remove that column. If the migration added an index, then rollback will remove that index.

> There is also a command for rolling back the previous migration and running it. It is `rails db:redo` .

Rails is smart enough to know how to reverse most of the migrations. But we can also provide hints to Rails on how to revert a migration by providing `up` and `down` methods instead of using the `change` method. The `up` method will be used when the migration is run whereas the `down` method will be used when the migration is rolled back.

```
def up
  change_table :events do |t|
    t.change :price, :string
  end
end

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

In this example, we are changing the `price` column of `events` from `integer` to `string` . We specify how it should be rolled back in the `down` method.

This same migration can also be written using the `change` method.

```ruby
def change
  reversible do |direction|
    change_table :events do |t|
      direction.up { t.change :price, :string }
      direction.down { t.change :price, :integer }
    end
  end
end
```

Rails also provides a way to revert a previous migration completely using the `revert` method.

```ruby
def change
  revert CreateEvents

  create_table :events do
   ...
  end
end
```

The `revert` method also accepts a block to revert a migration partially.

```ruby
def change
  revert do
    reversible do |direction|
      change_table :events do |t|
        direction.up { t.remove :event_type }
        direction.down { t.string :event_type }
      end
    end
  end
end
```

Executing It Raw

Sometimes, we want to execute complex SQL inside a migration. In such cases, we can forget the typical migration DSL and instead execute raw SQL as follows.

```ruby
def change
  execute <<-SQL
    ....
  SQL
end
```

Multiple Databases and Migrations

Rails 6 added support for using multiple databases within a single Rails application. If we want to use multiple databases, we configure them in the `database.yml` file.

```yaml
development:
  primary:
    <<: *default
    database: db/development.sqlite3
  analytics:
    adapter: sqlite3
    database: db/analytics_dev.sqlite3
```

This configuration tells Rails that we want to use two databases— `primary` and `analytics` . As we saw earlier, the migrations are stored in the `db/migrate` directory by default. But in this case, we can't add migrations of both databases inside a single directory. We don't want to run migrations of the `analytics` database on the `primary` database and vice versa. If we are using multiple databases, we are required to provide a path for storing migrations for the second database. This can be done by providing a `migrations_paths` in the `database.yml` .

```yaml
development:
  primary:
    <<: *default
    database: db/development.sqlite3
  analytics:
    adapter: sqlite3
    database: db/analytics_dev.sqlite3
    migrations_paths: db/analytics_migrate
```

We can then create migrations for the `analytics` database as follows.

```
rails generate migration AddExperiments rule:string active:boolean --db=analytics
```

This will create the migration inside `db/analytics_migrate` , and we can run it as follows.

```
rails db:migrate --db=analytics
```

If we only run the `rails db:migrate` , it will execute migrations for all the databases.

> The `analytics` database will have its own `schema_migrations` table to keep track of which migrations are run and which are not.

Running Migrations During Deployment

Since migrations can change the state of the database, and our code might depend on those changes, it is extremely important that the migrations are run first before the new code is applied.

In Heroku based deployments, migrations can be run in the `release` phase of the `Procfile` .

```
# Profile
web: bin/puma -C config/puma.rb
release: bundle exec rake db:migrate
```

This ensures that the migrations are run before the app dynos are restarted.

In Capistrano based deployments, migrations should run before the server is restarted.

In docker based deployments, we can run a sidecar container to run the migrations first before the app is restarted. This is very important as otherwise, the new containers can go into an inconsistent state if they start using new code before applying the database changes for that new code.

Conclusion

In this post, we saw various aspects of writing a database migration in Rails. We also saw what constitutes a migration as well as how to handle failures and roll back the migrations if needed. Rails 6 allows us to use multiple databases and the migrations for each need to be added separately. Finally, we briefly saw how to run the migrations during deployment so that database changes are applied properly before any new code starts using them.

P.S. If you'd like to read Ruby Magic posts as soon as they get off the press, subscribe to our Ruby Magic newsletter and never miss a single post!

Guest author Prathamesh Sonpatki is a developer working in Ruby and Ruby on Rails. He also co-organizes RubyConfIndia and DeccanRubyConf.