
Ruby on Rails

— Introducción a Migrations —

Overview

- Migrations
- Motivación del uso de migrations
- Cómo funcionan

Migrations / Motivación: agilidad

- La agilidad dentro de las aplicaciones es una gran ventaja:
 - *Lo único constante en los requerimientos de software es que siempre están cambiando.*
- Pero cómo monitoreamos y retrocedemos cambios en la base de datos?
- No existe una forma sencilla - aplicar y retroceder los cambios manualmente es un desastre además de propenso a errores.

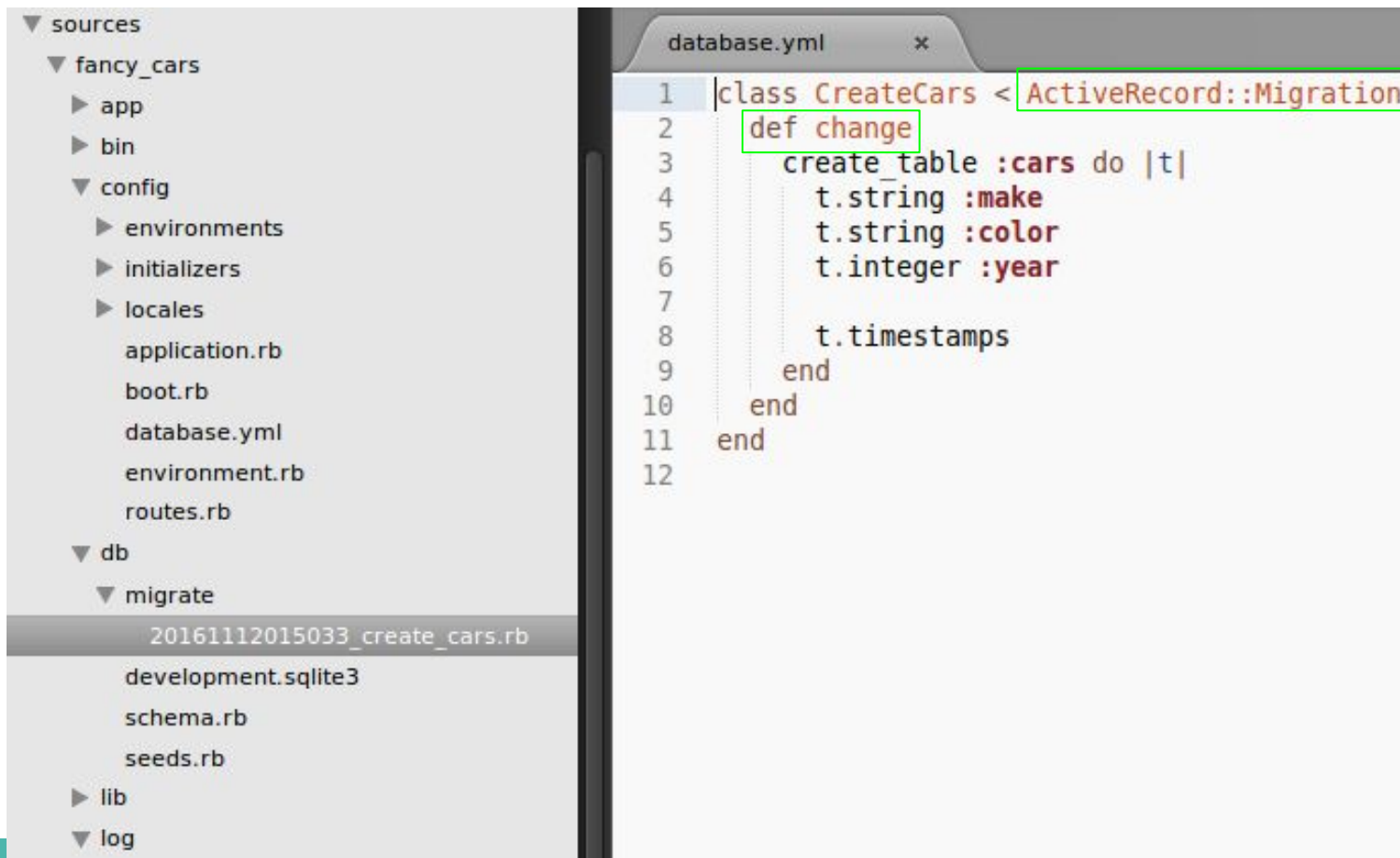
Migrations: Motivación Intercambio de BD

- Típicamente, SQL (o más específicamente DDL) es utilizado para **crear** y **modificar** tablas para una determinada base de datos **relacional**.
- Esto parece intuitivo, pero qué pasa si tenemos que cambiar de bases de datos en el medio del desarrollo?

Introducción a migrations

- Son clases ruby que extienden de ActiveRecord::Migration
- Básicamente el **nombre** del archivo debe empezar con un **timestamp** (year, month, date, hour, minuto, second) y ser seguido de algún nombre que es el nombre de la **clase**.
- El **timestamp** define una secuencia de cómo serán aplicados los cambios en la base de datos, y a veces actúa como una base de datos de versiones ordenadas o snapshots en el tiempo.

Migrations



The image shows a code editor interface with a file explorer on the left and a code editor on the right. The file explorer shows a project structure with folders like 'sources', 'fancy_cars', 'config', 'db', and 'log'. The 'db' folder is expanded, showing a 'migrate' subfolder. The 'migrate' folder contains a file named '20161112015033_create_cars.rb'. The code editor on the right shows the content of this file, which is a Ruby migration class. The class is named 'CreateCars' and inherits from 'ActiveRecord::Migration'. It has a 'change' method that defines a table named 'cars' with columns 'make', 'color', and 'year', and a 'timestamps' column. The code is highlighted with green boxes around the class name and the 'change' method definition.

```
▼ sources
  ▼ fancy_cars
    ► app
    ► bin
    ▼ config
      ► environments
      ► initializers
      ► locales
      application.rb
      boot.rb
      database.yml
      environment.rb
      routes.rb
    ▼ db
      ▼ migrate
        20161112015033_create_cars.rb
        development.sqlite3
        schema.rb
        seeds.rb
    ► lib
    ▼ log
```

```
database.yml x
1 class CreateCars < ActiveRecord::Migration
2   def change
3     create_table :cars do |t|
4       t.string :make
5       t.string :color
6       t.integer :year
7
8       t.timestamps
9     end
10  end
11 end
12
```

Creando migrations

- Se pueden **crear** migrations manualmnete, pero es menos propenso a errores utilizar el generador.
- Ya vimos el scaffold puede generar un migration (a no ser que se pase un parametro --no-migration)
- Existe una forma de crear un migration explícitamente con un Migration Generator.

Aplicando los migrations

- Una vez que se crea el **migration** (manualmente o mediante el generador), se necesita **aplicarlo** a la base con el objetivo de “**migrar**” la base de datos a su nuevo **estado**.
- No pueden haber **dos** migrations con el mismo **nombre de clase**.
- Al correr **rake db:migrate** se aplican todos los migrations en db/migrate que tengan un timestamp menor a la última vez que se corrió el comando.

Cómo funcionan los migrations

- Internamente está codificado para mantener una **tabla** llamada **schema_migrations** con una única columna llamada **version**.
- Una vez que los migrations son aplicados mediante rake db:migrate, la **version**(timestamp) va dentro de **schema_migrations**.
- En consecuencia, si corremos db:migrate (sobre los mismos migrations) varias veces no tendrá ningún efecto.

Anatomía de un Migration

- Qué va **dentro** de una subclase de ActiveRecord::Migration?
- Cualquiera de los siguientes:
 - def **up**
 - **Genera** los cambios de **schema**
 - def **down**
 - **Deshace** los cambios introducidos por el método **up**.
 - def **change**
 - Rails es capaz de saber cómo deshacer los cambios realizados en el método (casi siempre).

Independencia de base de datos

- En lugar de especificar tipos de datos específicos de una base de datos / los migrations nos permiten especificar tipos de datos lógicos.
- El adapter de ruby para la base de datos que estamos utilizando, realiza la traducción al tipo de datos para nuestra base de datos.
 - Entonces, para MySQL - podría terminar siendo un tipo de datos, para Postgres otro.

Mapeo de tipos de datos

Migration type	Sqlite3	Oracle	Postgres
:binary	blob	blob	bytea
:boolean	tinyint(1)	number(1)	boolean
:date	date	date	date
:datetime	datetime	date	timestamp
:decimal	decimal	decimal	decimal
:float	float	number	float
:integer	Integer	number(38)	integer
:string	varchar(255)	varchar2(255)	character varying
:text	text	clob	text
:time	datetime	date	time

Otras especificaciones sobre las columnas

- Más allá de especificar el tipo de datos, se pueden especificar **tres** opciones más (siempre que la base de datos las soporte).
- **null: true o false**
 - Cuando es false - se agrega un constraint **not null**
- **limit: size**
 - Establece un **límite** de tamaño para el campo. Si es un string se puede decir la cantidad de caracteres.
- **default: value**
 - Valor por defecto de la columna.
 - Se calcula una sola vez, cuando se corre el migration.

Opciones para columnas decimales

- Las columnas decimales pueden tomar dos opciones mas
- **precision: value**
 - La cantidad total de dígitos a ser almacenados
- **scale: value**
 - Dónde se pone el punto decimal.
 - Por ejemplo, un precision de 5 y un scale de 2 puede almacenar valores en el rango -999.99 y +999.99

Entonces

- Los migrations son clases Ruby que son traducidas al lenguaje de la base de datos.
- Una tabla en la base de datos **hace el seguimiento** de qué migration se aplicó la última vez. (rake db:migrate, rake db:rollback)