Databases and Migrations

COSC 480
21 February 2018

Joel Sommers
jsommers@colgate.edu
Colgate University

The Database is Gold

- Contains valuable customer data — don't want to test your app on the *real* data!
- Rails solution: development, production, and test *environments* each have their own DB
  - Different DB types appropriate for each
  - E.g., simple file-based database for test/development, like sqlite3
  - E.g., Industry-strength DB for production, like mysql, postgresql, or something else
- How to make changes to each DB, and maintain consistency across environments?
- Rails solution: migration
  - Migration: a script describing changes to tables in the database

Migration Advantages

- Can identify each migration and know which ones have been applied, and when
  - Migrations can often be designed to be reversible
- Can manage with version control
- Automated == reliably repeatable!
  - Compare: use bundler vs. manually installing libraries/gems
- Theme: don't do it manually, automate it!
  - Specify what to do, create tools to automate

Meet a code generator

`rails generate migration CreateProducts`

- Create the migration template
- Note: it is only created, not *applied* yet

- Apply migration to development DB: `rails db:migrate`
- A table describing which migrations have been applied is part of the database!

- Apply migration to production DB:

  `heroku run:detached rails db:migrate`

Applying migration also records in DB itself which migrations have been applied.

Adding a new or updating a) model to a Rails app

1. Create a migration describing the changes:
   `rails generate migration (name)`
   - Type `rails g migration` to get help
   - Many naming conventions to make migrations easy
2. Apply the migration: `rails db:migrate`
3. If new model (instead of modifying an existing a model)
   3.1 Create model file in `app/models/model.rb`
   3.2 Update test DB schema: `rails db:test:prepare`

Can also use the rails model generator

- `rails generate model`
- It also creates a migration script along with `app/models/model.rb`
  - Can specify column names/types to the model generator
  - And also to the migration generator

Checkpoint

What kind of object is likely being yielded in the migration code:

```
def up
  create_table :movies do |t|
    t.datetime 'release_date'
    # ...
  end
end
```

(A) An object representing a database

(B) An object representing an instance of a model

(C) An object representing a table

(D) Seriously, it could be anything

Checkpoint

What kind of object is likely being yielded in the migration code:

```
def up
  create_table :movies do |t|
    t.datetime 'release_date'
    # ...
  end
end
```

(A) An object representing a database

(B) An object representing an instance of a model

(C) An object representing a table

(D) Seriously, it could be anything

C — seriously, it's a table.

CRUD in SQL

- Structured Query Language (SQL) is the query language used by Relational Databases
- Rails generates SQL statements at runtime, based on Ruby code
- 4 basic operations on a table row: Create, Read, Update, Delete

The following is SQL (which we generally won't have to write)

```
INSERT INTO users(username, email, birthdate)
    VALUES ("joel", "jsommers@colgate.edu", "1972-06-10"),
    "halima", "----", "2008-12-20")
SELECT * FROM users
    WHERE (birthdate BETWEEN "1990-01-01" AND "2010-01-01")
UPDATE users SET email = "jsommers@acm.org" WHERE username="joel"
DELETE FROM users WHERE id=1
```

Experimenting and debugging with a model

- It is often useful to "play" with a model class and its method
- `rails console` creates an irb session with your model classes preloaded
- Can do any CRUD operation on a model
- Awesome for experimenting...
- Also useful for debugging...
- See the SQL generated by any method...

The Ruby side of a model

- Models are subclassed from `ApplicationRecord`
  - "Connects" a model to the database
  - Provides CRUD operations on the model
- Rails convention: database table name is derived from model's name
  - `Product` (model class) → `products` (table)
- Database table column names are getters and setters for model attributes
- Key: the getters and setters do not simply modify instance variables!

```
# 3 ways to create ActiveRecord objects
product = Product.new
product.name = 'Stuffed animal'
product.price = 11.99

product = Product.new do |p|
  p.name = 'Stuffed animal'
  p.price = 11.99
end

product = Product.new(:name => 'Stuffed animal', :price => 11.99)
```

Model attributes

- Model class provides a way to access database table row by manipulating Ruby object
- Table columns should generally be accessed using `self.attribute` within any instance methods of the model class
  - Why?

```
class Product < ApplicationRecord
   def capname
       self.name.capitalize
       # NOT @name.capitalize
   end
end
```

Creating a model object: new $\neq$ save

- Must call `save` or `save!` on an ActiveRecord model instance to actually save changes to DB
  - ! version is dangerous: throws an exception if operation fails
  - `create` method does both `new` and `save`
- Once created, object acquires a primary key (`id` column)
  - if `x.id` is `nil` or `x.new_record?` is true, x has never been saved
  - `x.persisted?` is another way to test whether an object has been saved to the database
  - These behaviors are inherited from `ActiveRecord::Base`

Read: finding things in the DB

`all` class method retrieves all objects

```
@products = Product.all
```

Other basic class methods for object retrieval:

- `first`: retrieves first object (lowest id)
- `last`: object with largest id

The `find` class method is used to find objects by id:

```
p = Product.find(13) # get object with id 13
products = Product.find([13,42]) # find understands arrays of ids
```

The `find_by` class method is used to find objects by attribute:

```
p = Product.find_by(name: "fuzzy bunny")
```

Read: where method

Class method `where` selects objects based on attributes

- Argument to `where` can be a string or a hash

```
Product.where('price > 2')
Product.where('price < ?', 2) # parameterized
Product.where('name like ?', "%#{matchstr}%")

# don't use Ruby string interpolation to fill in queries!
Product.where("price < #{limit}") # BAD!  Do NOT do this!
```

`where` can be chained

```
Product.where('price <= 5').where('name like ?', "%#{fuzzy}%")
```

- http://guides.rubyonrails.org/active_record_basics.html

Checkpoint

Consider the following example, in which interpolation is performed in an ActiveRecord query:

```
Product.where("price > #{minprice}")
```

Why is interpolation bad in ActiveRecord queries?

A. Interpolation doesn't work in this context; Ruby doesn't allow it.

B. Performance; interpolation is too slow.

C. It's a database security threat.

D. It's bad because you should use `find_by` instead of where.

Checkpoint

Consider the following example, in which interpolation is performed in an ActiveRecord query:

```
Product.where("price > #{minprice}")
```

Why is interpolation bad in ActiveRecord queries?

A. Interpolation doesn't work in this context; Ruby doesn't allow it.

B. Performance; interpolation is too slow.

C. It's a database security threat.

D. It's bad because you should use `find_by` instead of where.

C. Where did minprice come from? If it came from your application, then maybe it's ok. If it came (even indirectly) from the browser, this is bad, bad, bad.

Updating

- Modify attributes, then save object

```
p = Product.find(42)
p.price = p.price * 1.1 # markup
p.save! # raises exception if save fails
p.save  # returns true if saved, false otherwise
```

- Update attributes on existing object

```
Product.find(1).
  update(:price => 49.99) # accepts a hash
# or
Product.update(:id=>1, :price=>49.99)
```

- Update all matching records

```
# update_all takes hash of attributes to change or a string
Product.update_all("price = price * 0.90")
```

- Transactional: either all attributes are updated, or none are

Delete is straightforward

- Destroy is an instance method

```
p = Product.find(13)
p.destroy
```

- There's also `delete`, which does not trigger lifecycle callbacks (we'll discuss later)
  - Generally do not want delete; use destroy
- Once an AR object is destroyed, you can access but not modify the in-memory object

```
p.name = 'Slinky 2' # fails -
                    # object is 'frozen'
```

Checkpoint

Assume table `fortune_cookie` has column `fortune_text`. Assume also that we have a model `FortuneCookie < ApplicationRecord`. Which of these instance methods will not return a silly fortune (if any)?

(A)
```ruby
def silly_fortune_1
  @fortune_text + 'in bed'
end
```

(B)
```ruby
def silly_fortune_2
  self.fortune_text + 'in bed'
end
```

(C)
```ruby
def silly_fortune_3
  fortune_text + 'in bed'
end
```

(D) They will all return a silly fortune

Checkpoint

Assume table `fortune_cookie` has column `fortune_text`. Assume also that we have a model `FortuneCookie < ApplicationRecord`. Which of these instance methods will not return a silly fortune (if any)?

(A)
```ruby
def silly_fortune_1
  @fortune_text + 'in bed'
end
```

(B)
```ruby
def silly_fortune_2
  self.fortune_text + 'in bed'
end
```

(C)
```ruby
def silly_fortune_3
  fortune_text + 'in bed'
end
```

(D) They will all return a silly fortune

B or C are ok, but A is not. (Correct answer is A: it will not return a silly fortune.)

“Seeding” the database

- It’s often useful to “seed” the database to provide an initial set of model instances (by adding rows to db tables)
- Add to `db/seeds.rb`
  - Use any ActiveRecord calls to create/save model instances
  - Load seeds into db with `rails db:seed`

```ruby
# assuming decimal field for price
Product.create!(name: "fuzzy bunny", price: 11.99)
Product.create!(name: "slinky", price: 3.49)

# - or - make a hash of attributes, and use a each "loop"
products = [
    { name: "fuzzy bunny", price: 11.99 },
    { name: "slinky", price: 3.49 },
]
products.each do |pdata|
    Product.create!(pdata)
end
```

Summary: ActiveRecord

- Subclassing `ApplicationRecord` (which derives directly from `ActiveRecord::Base`) "connects" a model to the database
- C (save,create), R (all,first,last,where,find), U (update, update_all), D (destroy)
- Convention over configuration maps
  - model name to DB table name
  - getters/setters to DB table columns
- Object in memory $\neq$ row in database
  - `save` must be used to persist
  - `destroy` doesn't destroy in-memory copy