In this lecture, we will discuss…

- How to scaffold resources quickly
- What Rails scaffolding gives you out of the box

Scaffolding

- Essentially a code-generator for entities
- Generates a simple prototype to work with…
- Lets you get up and running quickly

Great for learning Rails!

Scaffolding Generator

```
~$ cd fancy_cars/
~/fancy_cars$ rails g scaffold car make color year:integer
      invoke  active_record
      create    db/migrate/20150907153643_create_cars.rb
      create    app/models/car.rb
      invoke    test_unit
      create      test/models/car_test.rb
      create      test/fixtures/cars.yml
      invoke  resource_route
       route    resources :cars
      invoke  scaffold_controller
      create    app/controllers/cars_controller.rb
      invoke    erb
      create      app/views/cars
      create      app/views/cars/index.html.erb
      create      app/views/cars/edit.html.erb
      create      app/views/cars/show.html.erb
      create      app/views/cars/new.html.erb
      create      app/views/cars/_form.html.erb
```

rails new fancy_cars

Resource (model) and column names and types

String assumed for type…

Applying Scaffolding to DB

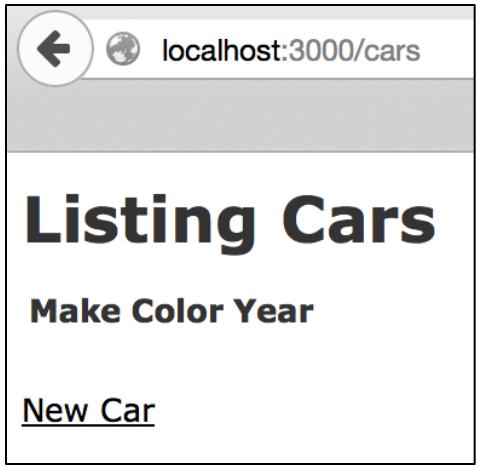
```
~/fancy_cars$ rake db:migrate
== 20150907153643 CreateCars: migrating ======
-- create_table(:cars)
   -> 0.0010s
== 20150907153643 CreateCars: migrated (0.0011s)
```

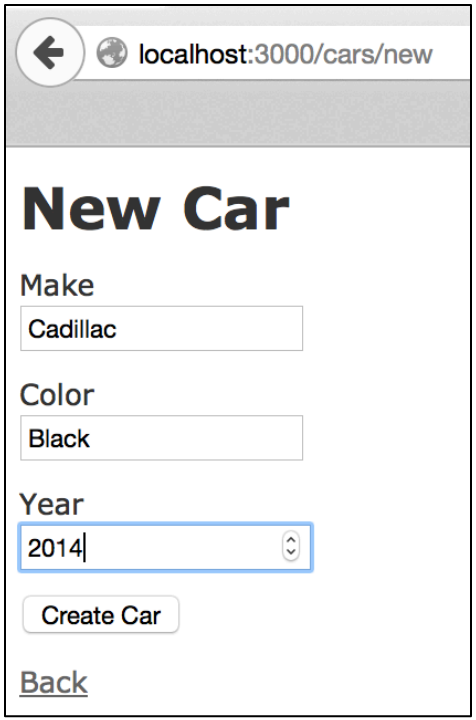
NOTE: This is a ***rake*** task, but the scaffold generation is a ***rails*** generation

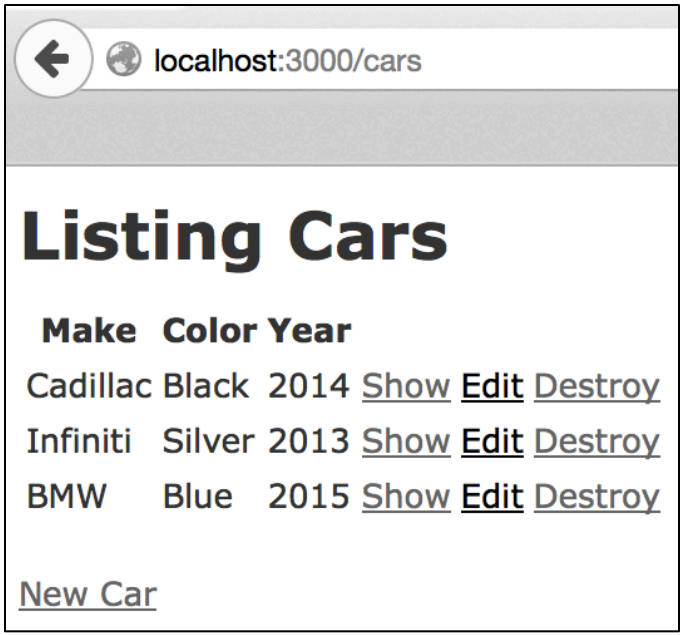
Apply scaffolding to the database, a.k.a. migrate the DB.

rails s or
rails server

Scaffolding Demo







Scaffolding – API also?

But wait…
there's more!

localhost:3000/cars.json


```
[
  ▼ {
        id: 1,
        make: "Cadillac",
        color: "Black",
        year: 2014,
        url: http://localhost:3000/cars/1.json
    },
  ▼ {
        id: 2,
        make: "Infiniti",
        color: "Silver",
        year: 2013,
        url: http://localhost:3000/cars/2.json
    },
  ▼ {
        id: 3,
        make: "BMW",
        color: "Blue",
        year: 2015,
        url: http://localhost:3000/cars/3.json
    }
]
```

Scaffolding – Explaining the Magic

- Scaffolding creates:
  1. *Migration*
  2. *Model*
  3. Routes
  4. Controller with actions
  5. Views

And more…

Summary

- ✧ Scaffolding can get you up and running quickly
- ✧ Generates JSON response

What's Next?

- ✧ Database Setup

In this lecture, we will discuss…

- ✧ Which database is Rails using?
- ✧ How to view the database

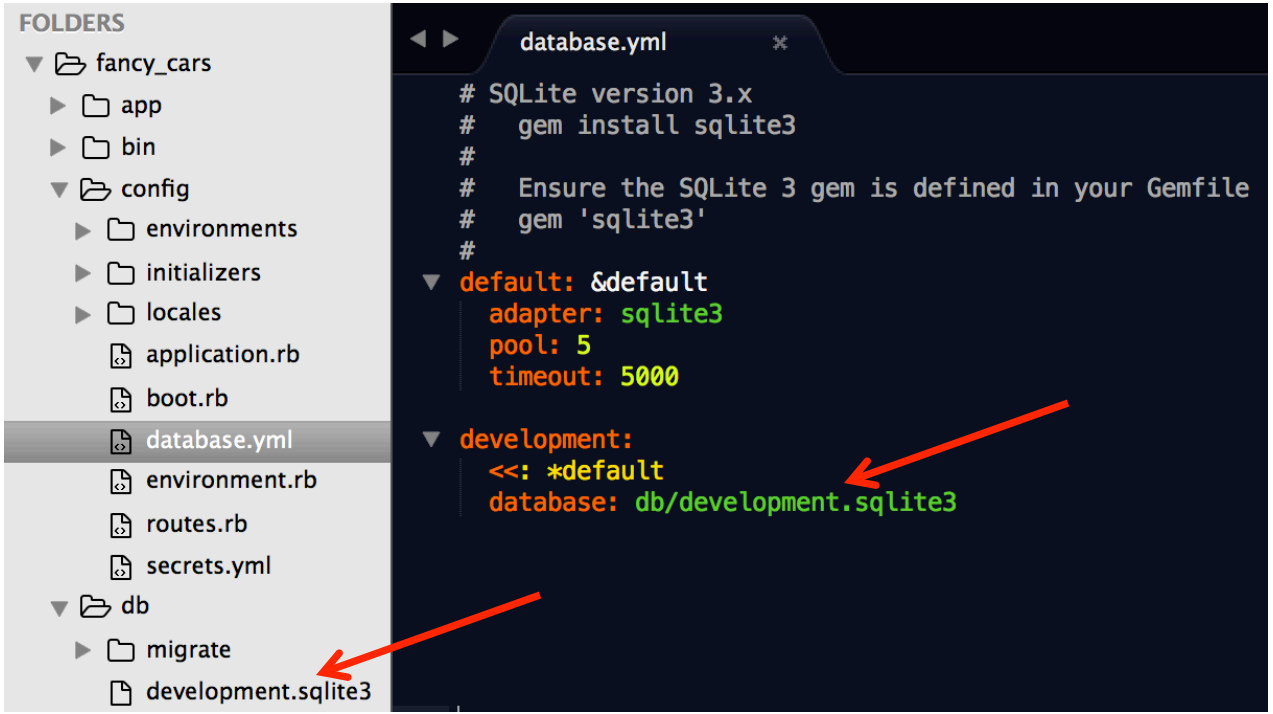
SQLite

- Rails uses SQLite for database by default
  - Self-contained, server-less, zero-configuration, transactional, relational SQL database engine

CLAIM: Most widely deployed SQL database engine in the world

Database Setup: config/database.yml

- Which database are we using?



Database Console: rails db

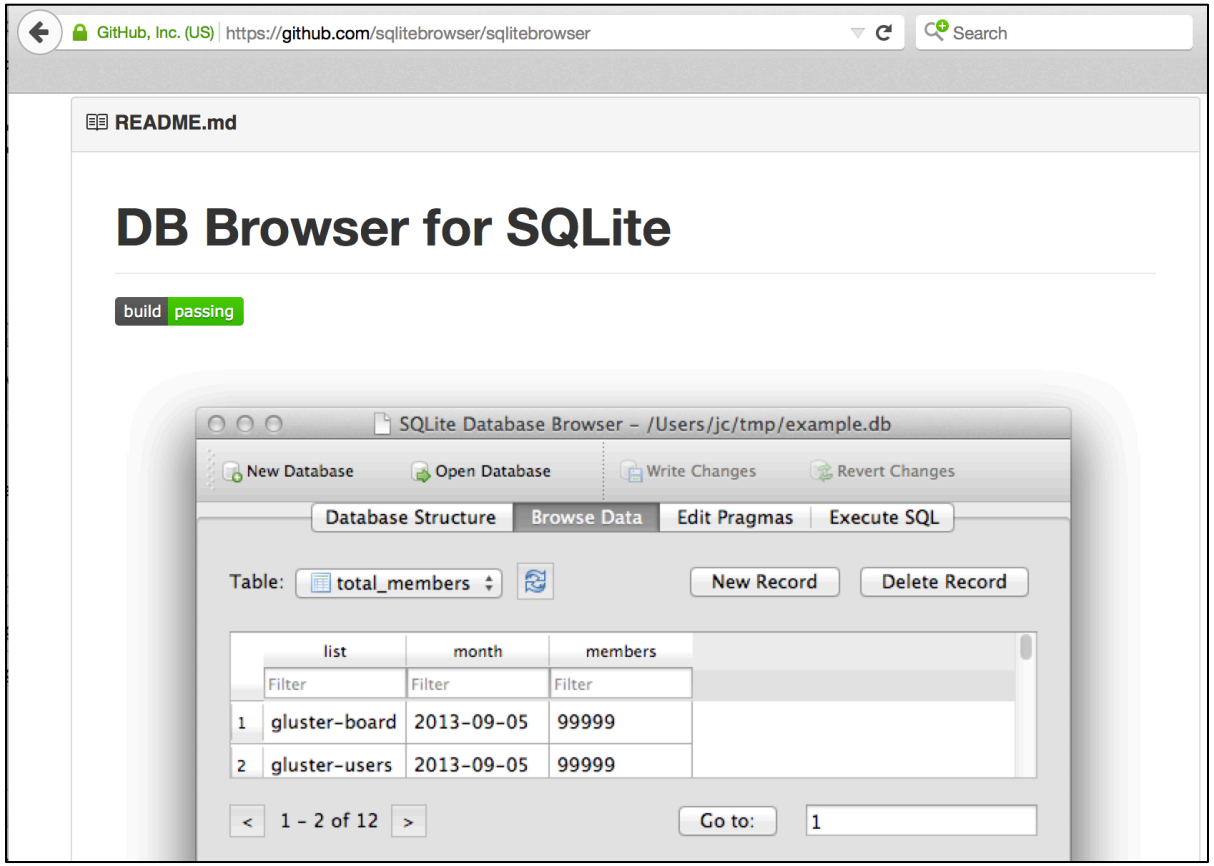
```
~/fancy_cars$ rails db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .help
.backup ?DB? FILE      Backup DB (default "main") to FILE
.bail on|off           Stop after hitting an error.  Default OFF
.clone NEWDB           Clone data into NEWDB from the existing database
.databases             List names and files of attached databases
.dump ?TABLE? ...      Dump the database in an SQL text format
                         If TABLE specified, only dump tables matching
                         LIKE pattern TABLE.
.echo on|off           Turn command echo on or off
.exit                  Exit this program
.explain ?on|off?      Turn output mode suitable for EXPLAIN on or off.
                         With no args, it turns EXPLAIN on.
.headers on|off        Turn display of headers on or off
.help                  Show this message
.import FILE TABLE     Import data from FILE into TABLE
.indices ?TABLE?       Show names of all indices
```

Some useful SQLite options

- Good idea: turn headers on and column mode

```
~/fancy_cars$ rails db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .tables
cars               schema_migrations
sqlite> .headers on
sqlite> .mode columns
sqlite> select * from cars;
id          make        color       year        created_at                  updated_at
----------  ----------  ----------  ----------  --------------------------  --------------------------
1           Cadillac    Black       2014        2015-09-07 16:48:07.474368  2015-09-07 16:48:07.474368
2           Infiniti    Silver      2013        2015-09-07 16:48:33.290148  2015-09-07 16:48:33.290148
3           BMW         Blue        2015        2015-09-07 16:48:49.017993  2015-09-07 16:48:49.017993
sqlite> .exit
```

Other Tools for Viewing SQLite



Summary

- ✧ Rails uses SQLite by default
- ✧ Built-in command-line DB viewer

What's Next?

- ✧ Introduction to Migrations

In this lecture, we will discuss…

- Migrations
- Incentives for migrations
- How migrations work

Migrations – Agility Incentive

- Agility inside the applications is a given!
  - *"The only constant thing about software requirements is that they are always changing"*
- But how do we monitor and undo changes to the DB?
- There is no easy way – manually applying and undoing changes is messy and error-prone.

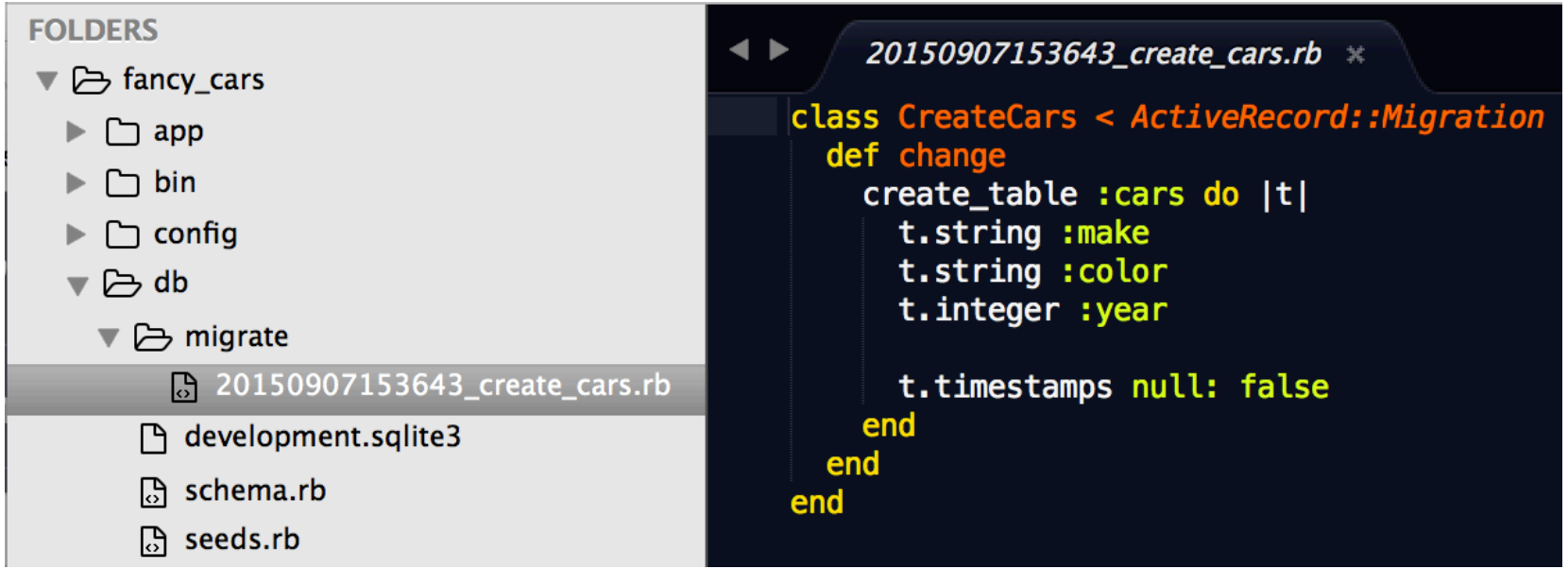
Migrations: Cross Database Incentive

- Typically, SQL (or more specifically DDL) is used to create and modify tables for a particular relational database
- This feels intuitive, but what if you have to switch databases in the middle?
  - For example, develop on SQLite and deploy to Postgres

Enter Migrations

- Ruby `classes` that extend `ActiveRecord::Migration`
- File name needs to start with a timestamp (year/month/date/hour/minute/second) and be followed by some name, which becomes the name of the `class`
- This timestamp defines the sequence of how the migrations are applied and acts as a database version of sorts or snapshot in time

Migrations


FOLDERS
fancy_cars
app
bin
config
db
migrate
20150907153643_create_cars.rb
development.sqlite3
schema.rb
seeds.rb
20150907153643_create_cars.rb
class CreateCars < ActiveRecord::Migration
def change
create_table :cars do |t|
t.string :make
t.string :color
t.integer :year
t.timestamps null: false
end
end
end

Creating Migrations

- You can create migrations by hand, but it's obviously much less error-prone to use a generator
- We already saw that scaffold generator creates a migration (unless passed `--no-migration` flag)
- There is also an explicit migration generator

Applying Migrations

- Once the migration is created (either manually or through a generator) it needs to be applied to a database in order to “migrate” the database to its new state
- No two migrations can have the same `class` name
- You run `rake db:migrate` to apply all migrations in `db/migrate` folder in (timestamp) order

How Do Migrations Work?

- Migration code maintains a table called `schema_migrations` table with one column called `version`
- Once the migration is applied – its version (timestamp) goes into the `schema_migrations` table
- It therefore follows that running `db:migrate` (on the same set of migrations) multiple times will have no effect

Anatomy of Migration

- So, what actually goes inside the `ActiveRecord::Migration` subclass?
- Either
  - `def up`
    - Generate db schema changes
  - `def down`
    - Undo the changes introduced by the `up` method
- Or, just `change` method when Rails can guess how to undo changes (most of the time)

Database Independence

- Instead of specifying database-specific types for particular DB type – migrations let you specify logical database types
- The ruby database adapter you are using does the translation to the actual DB type.
  - So, for MySQL – it will end up being one type, Postgres another and so on.

Actual Type Mapping

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

Extra Column Options

- Besides specifying logical types, you can specify up to three more options (when the underlying DB supports it)
- `null: true` or `false`
  - When `false` – a `not null` constraint is added
- `limit: size`
  - Sets a limit on the size of the field
- `default: value`
  - Default value for the column (Calculated once!)

Decimal Column Options

- Decimal columns (optionally) take two more options
- `precision: value`
  - Total number of digits stored
- `scale: value`
  - Where to put the decimal point
    - For example, precision 5 and scale 2 can store the range -999.99 to 999.99

Summary

- ✧ Migrations are just Ruby classes that get translated into DB speak
- ✧ Table in the DB keeps track of which migration was applied last

What's Next?

- ✧ Creating and altering tables and columns

In this lecture, we will discuss…

- How to create and drop tables
- How to modify and rename columns

Creating Tables with Migrations

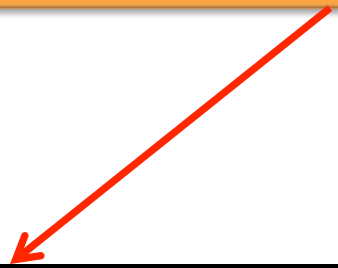
- By convention, table names in Rails are always named plural (many rows…)
- An `id` column is automatically created to be used as primary key
- `timestamps` method creates `created_at` and `updated_at` columns
- `create_table` and `drop_table` create and drop the table

Creating Cars Table

```ruby
class CreateCars < ActiveRecord::Migration
  def change
    create_table :cars do |t|
      t.string :make
      t.string :color
      t.integer :year

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

After `rake db:migrate`



```
sqlite> .schema cars
CREATE TABLE "cars" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, "make" varchar, "color"
 varchar, "year" integer, "created_at" datetime NOT NULL, "updated_at" datetime NOT NULL);
sqlite>
```

rake db:rollback

- Undoes the last migration (possibly applies the `down` method)

```
~/fancy_cars$ rake db:rollback
== 20150907153643 CreateCars: reverting ======
-- drop_table(:cars)
   -> 0.0011s
== 20150907153643 CreateCars: reverted (0.0055s)
```

Oh no!!! Where did my car data go? What data?...

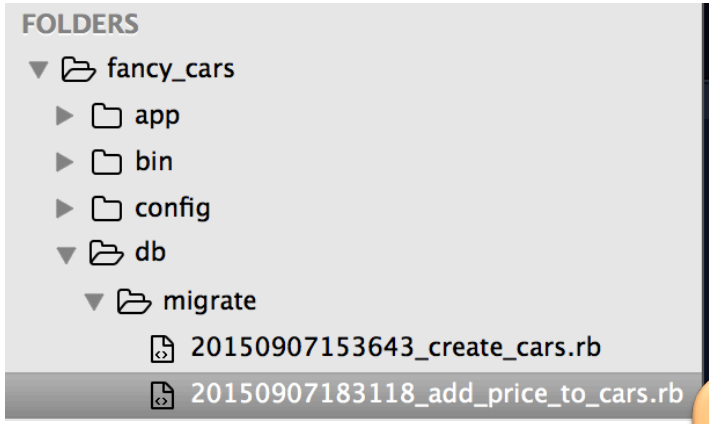
Adding / Removing Column

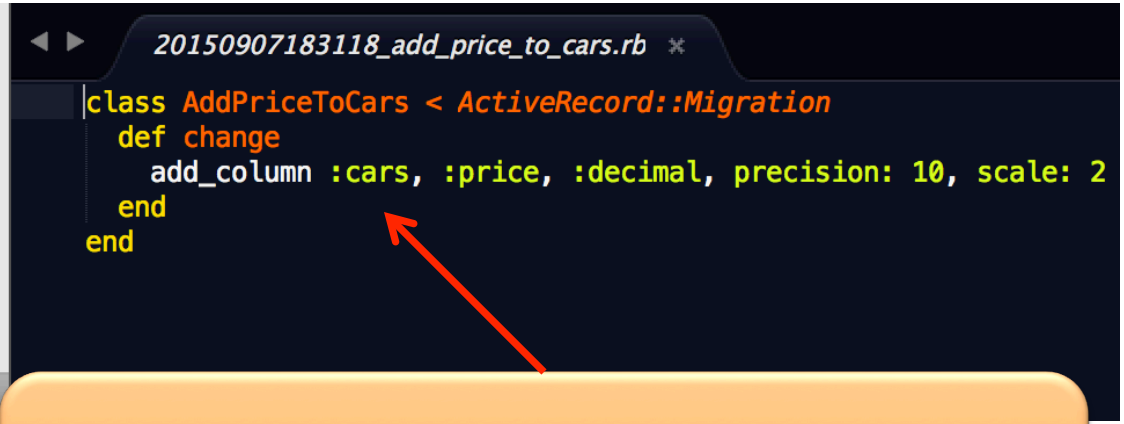
- `add_column :table_name, :column_name, :column_type`
- `remove_column :table_name, :column_name`

Let's add a price column to the cars table

Adding Price To Cars Table

```
~/fancy_cars$ rails g migration add_price_to_cars 'price:decimal{10,2}'
      invoke  active_record
      create    db/migrate/20150907183118_add_price_to_cars.rb
```





Automatically detects table name from how we named the migration

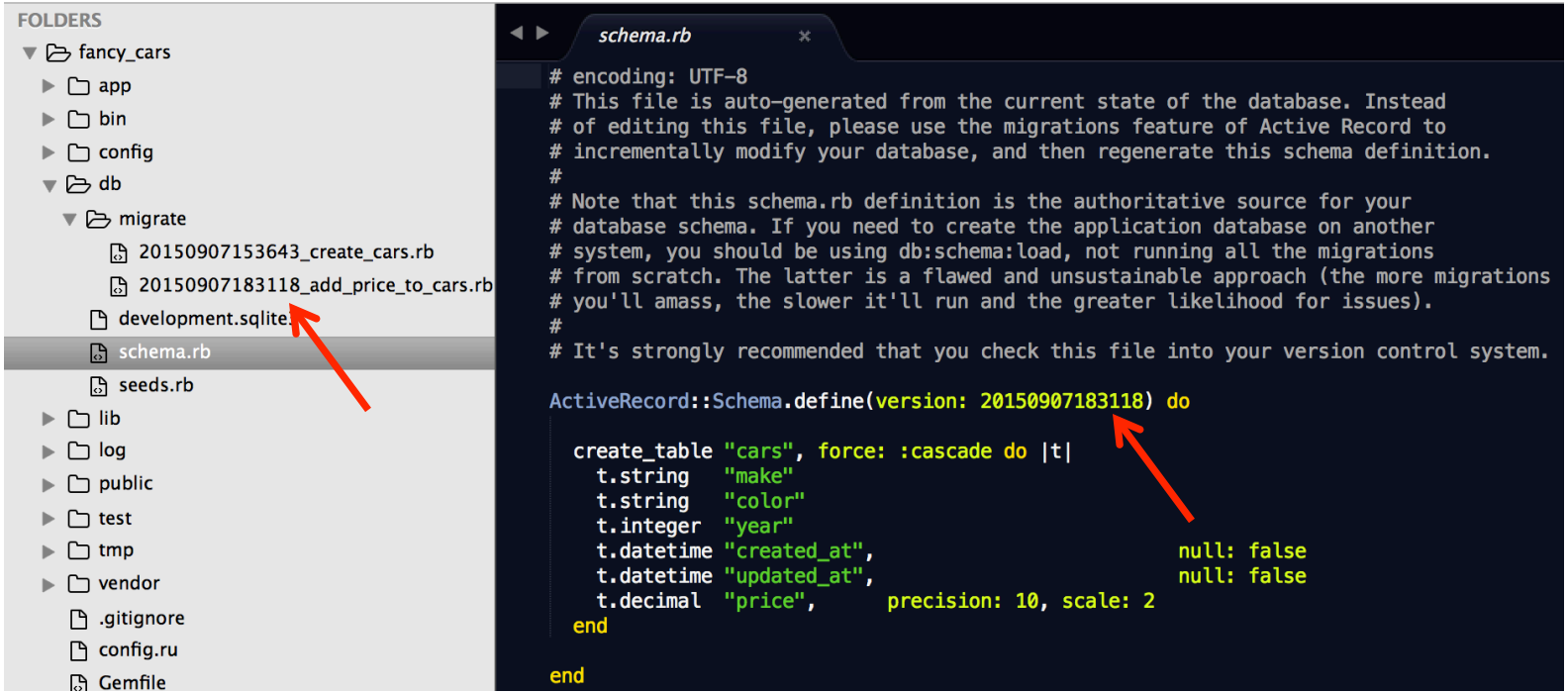
Adding Price To Cars Table

```
~/fancy_cars$ rake db:migrate
== 20150907153643 CreateCars: migrating ======================================
-- create_table(:cars)
   -> 0.0010s
== 20150907153643 CreateCars: migrated (0.0011s) =============================

== 20150907183118 AddPriceToCars: migrating ==================================
-- add_column(:cars, :price, :decimal, {:precision=>10, :scale=>2})
   -> 0.0005s
== 20150907183118 AddPriceToCars: migrated (0.0006s) =========================

~/fancy_cars$ rails db
SQLite version 3.8.5 2014-08-15 22:37:57
Enter ".help" for usage hints.
sqlite> .schema cars
CREATE TABLE "cars" ("id" INTEGER PRIMARY KEY AUTOINCREMENT NOT NULL, "make" varchar, "color" varchar, "
year" integer, "created_at" datetime NOT NULL, "updated_at" datetime NOT NULL, "price" decimal(10,2));
```

Overall Schema Of Your Database

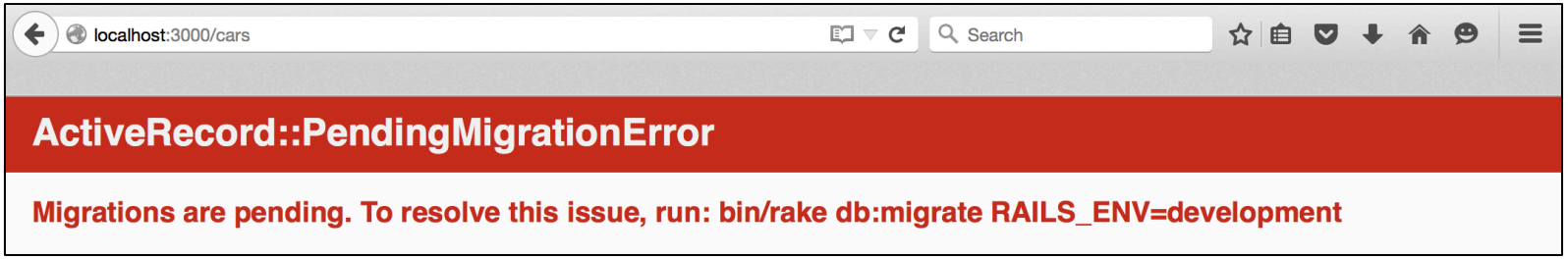


Renaming Columns

- `rename_column :table_name, :old_column_name, :new_column_name`

```
~/fancy_cars$ rails g migration rename_make_to_company
      invoke  active_record
      create    db/migrate/20150907185017_rename_make_to_company.rb
```

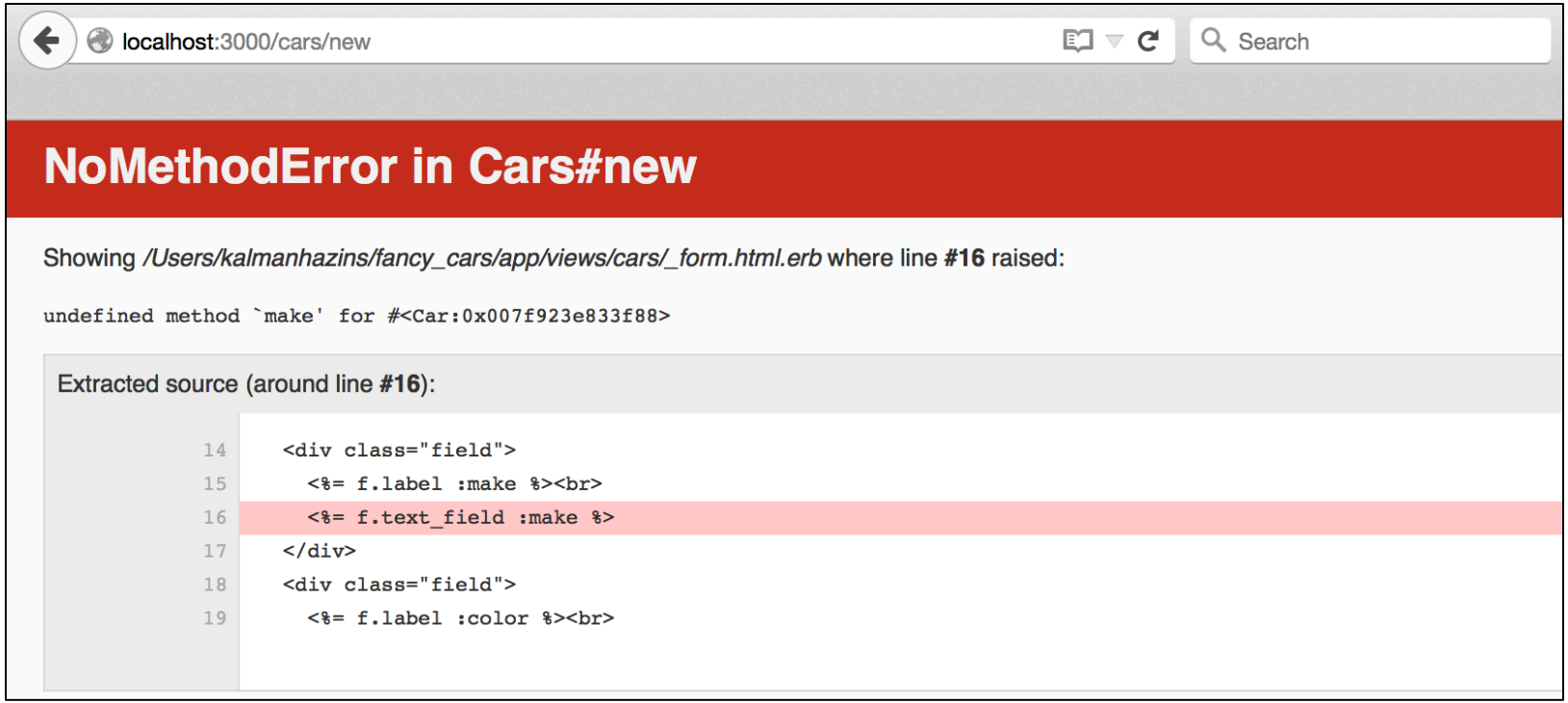
Renaming Columns




```
~/fancy_cars$ rake db:migrate
== 20150907185017 RenameMakeToCompany: migrating ==========
-- rename_column(:cars, :make, :company)
   -> 0.0036s
== 20150907185017 RenameMakeToCompany: migrated (0.0037s)
```

Does not destroy data – just renames the column…

Renaming Columns



Irreversible Migrations And More

- Sometimes, you only want your migrations to go one way (for example, when loss of data is unacceptable)
- Or, you run across any other issues related to migrations not discussed thus far
- Hit the rails guides before hitting Stack Overflow

http://guides.rubyonrails.org/migrations.html

Summary

- ✧ Use Rails generators to generate migrations
- ✧ Beware of side effects to your app due to migrations

What's Next?

- ✧ Dynamic Dispatch

In this lecture, we will discuss…

- The advantages and disadvantages of Ruby being a dynamic language
- Dynamic Dispatch

Dynamic

- In static languages, like Java, the compiler requires you to define all the methods upfront
- In dynamic languages, such as Python and Ruby, methods don't have to be predefined - they need to only be "found" when invoked
- Disadvantage?
  - Compiler can find bugs easier

Reporting System Example

- Say you have a `Store` class
  - Description and price of store products
- You are tasked with building a reporting system that can generate reports for different items in the store

Reporting System Example

```
class Store
  def get_piano_desc
    "Excellent piano"
  end
  def get_piano_price
    120.00
  end
  def get_violin_desc
    "Fantastic violin"
  end
  def get_violin_price
    110.00
  end

  # ...many other similar methods...
end
```

Reporting System Example

```
require_relative 'store'
class ReportingSystem
  def initialize
    @store = Store.new
  end
  def get_piano_desc
    @store.get_piano_desc
  end
  def get_piano_price
    @store.get_piano_price
  end

  # ...many more simimlar methods...
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

Calling Methods Dynamically

- So far, we have seen how to call methods using the dot notation `obj.method`
- It turns out, there is another way to call a method in Ruby - using the `send` method
- First parameter is the method name/symbol; the rest (if any) are method arguments
- Send?
  - Think of it as sending a message to an object

Calling Methods Dynamically

```ruby
class Dog
  def bark
    puts "Woof, woof!"
  end
  def greet(greeting)
    puts greeting
  end
end

dog = Dog.new
dog.bark # => Woof, woof!
dog.send("bark") # => Woof, woof!
dog.send(:bark) # => Woof, woof!
method_name = :bark
dog.send method_name # => Woof, woof!

dog.send(:greet, "hello") # => hello
```

Dynamic Dispatch: Advantages

- Advantages to dynamic method calling, a.k.a. "*Dynamic Dispatch*"
  - Can decide at runtime which methods to call
- The code doesn't have to find out until runtime which method it needs to call

Dynamic Dispatch Example

```
~$ irb
irb(main):001:0> props = { name: "John", age: 15 }
=> {:name=>"John", :age=>15}
irb(main):002:0> class Person; attr_accessor :name, :age; end
=> nil
irb(main):003:0> person = Person.new
=> #<Person:0x007f8d1c24b908>
irb(main):004:0> props.each { |key, value| person.send("#{key}=", value) }
=> {:name=>"John", :age=>15}
irb(main):005:0> person
=> #<Person:0x007f8d1c24b908 @name="John", @age=15>
```

Summary

- ✧ Don’t need to call a method using the dot notation
- ✧ Can call methods dynamically using a string or symbol

What’s Next?

- ✧ Dynamic Methods

In this lecture, we will discuss…

- ✧ Defining methods dynamically

Defining Methods Dynamically

- A.k.a. "*Dynamic Method*"
- Not only can you call methods dynamically (with `send`) - you can also define methods dynamically
- `define_method :method_name` and a block which contains the method definition
- Defines an instance method for the class

Dynamic Method Example

```
class Whatever
  define_method :make_it_up do
    puts "Whatever..."
  end
end

whatever = Whatever.new
whatever.make_it_up # => Whatever...
```

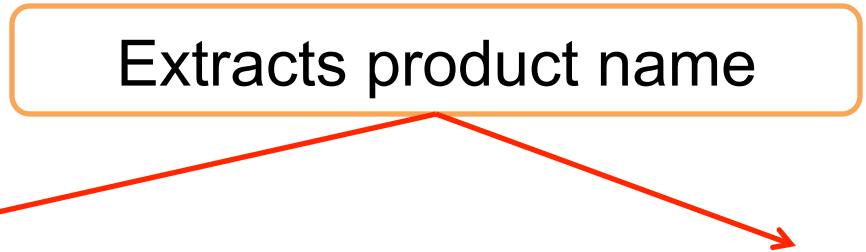
So Now, Instead of This…

```
require_relative 'store'
class ReportingSystem
  def initialize
    @store = Store.new
  end
  def get_piano_desc
    @store.get_piano_desc
  end
  def get_piano_price
    @store.get_piano_price
  end

  # ...many more similar methods...
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

…We Can Do This!




```
require_relative 'store'
class ReportingSystem

  def initialize
    @store = Store.new
    @store.methods.grep(/^get_(.*)_desc/) { ReportingSystem.define_report_methods_for $1 }
  end

  def self.define_report_methods_for (item)
    define_method("get_#{item}_desc") { @store.send("get_#{item}_desc")}
    define_method("get_#{item}_price") { @store.send("get_#{item}_price")}
  end
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

Improved Reporting System

- ✧ No more duplication
  - Now, you don’t have to write all of those repetitive methods anymore
- ✧ **Bonus:** If someone adds a new item to the `Store` class - your `ReportingSystem` class already “knows about it” (as long as the same method naming pattern is adhered to)

Summary

- ✧ Defining methods dynamically can dramatically reduce the amount of code that needs to be written

What's Next?

- ✧ Ghost methods

In this lecture, we will discuss…

- ✧ Ghost methods

Nonexistent (Ghost) Methods

- **Question:** If a method is invoked and it's not found, was it really called at all?

```
irb(main):001:0> class SomeClass; end
=> nil
irb(main):002:0> some_class = SomeClass.new
=> #<SomeClass:0x007fb552ae1c80>
irb(main):003:0> some_class.i_dont_exist
NoMethodError: undefined method `i_dont_exist' for #<SomeClass:0x007fb552ae1c80>
        from (irb):3
        from /Users/kalmanhazins/.rbenv/versions/2.1.2/bin/irb:11:in `<main>'
```

method_missing… method

- Ruby looks for the method invoked in the class to which it belongs
- Then it goes up the ancestors tree (classes and modules)
- If it still doesn't find the method, it calls `method_missing` method
- The default `method_missing` implementation throws `NoMethodError`

Overriding method_missing

- Since `method_missing` is just a method, you can easily override it
- You have access to
  - Name of the method called
  - Any arguments passed in
  - A block if it was passed in

Overriding method_missing

```ruby
class Mystery
  # no_methods defined
  def method_missing (method, *args)
    puts "Looking for..."
    puts "\"#{method}\" with params (#{args.join(',')}) ?"
    puts "Sorry... He is on vacation..."
    yield "Ended up in method_missing" if block_given?
  end
end

m = Mystery.new
m.solve_mystery("abc", 123123) do |answer|
  puts "And the answer is: #{answer}"
end

# => Looking for...
# => "solve_mystery" with params (abc,123123) ?
# => Sorry... He is on vacation...
# => And the answer is: Ended up in method_missing
```

Ghost Methods

- `method_missing` gives you the power to "fake" the methods
- Called "ghost methods" because the methods don't really exist
- Ruby's built-in classes use `method_missing` and dynamic methods all over the place…

Struct and OpenStruct

- `Struct`
  - Generator of specific classes, each one of which is defined to hold a set of variables and their accessors (“Dynamic Methods”)
- `OpenStruct`
  - Object (similar to `Struct`) whose attributes are created dynamically when first assigned (“Ghost methods”)

Struct and OpenStruct

```ruby
Customer = Struct.new(:name, :address) do # block is optional
  def to_s
    "#{name} lives at #{address}"
  end
end
jim = Customer.new("Jim", "-1000 Wall Street")
puts jim # => Jim lives at -1000 Wall Street

require 'ostruct' # => need to require ostruct for OpenStruct

some_obj = OpenStruct.new(name: "Joe", age: 15)
some_obj.sure = "three"
some_obj.really = "yes, it is true"
some_obj.not_only_strings = 10
puts "#{some_obj.name} #{some_obj.age} #{some_obj.really}"
# => Joe 15 yes, it is true
```

So Now, Instead Of This…

```
require_relative 'store'
class ReportingSystem
  def initialize
    @store = Store.new
  end
  def get_piano_desc
    @store.get_piano_desc
  end
  def get_piano_price
    @store.get_piano_price
  end

  # ...many more simimlar methods...
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

…We Can Do This!

```
require_relative 'store'

class ReportingSystem
  def initialize
    @store = Store.new
  end
  def method_missing(name, *args)
    super unless @store.respond_to?(name)
    @store.send(name)
  end
end

rs = ReportingSystem.new
puts "#{rs.get_piano_desc} costs #{rs.get_piano_price.to_s.ljust(6, '0')}"
# => Excellent piano costs 120.00
```

Why do we care to use "super" here?

method_missing and Performance

- Since the invocation is indirect, could be a little slower
- Most of the time, it will probably not matter too much
- If it does, you can consider a hybrid approach
  - Define a real method from inside `method_missing` after an attempted “call”

Summary

- ✧ Ghost methods allow you to call methods as if they are there even though they are not
- ✧ Method behavior can be defined at runtime, for example based on database columns existing or not!

What's Next?

- ✧ Introduction to Active Record

In this lecture, we will discuss…

- ✧ Active Record, the ORM
- ✧ What conventions does Active Record assume?

Models: ORM

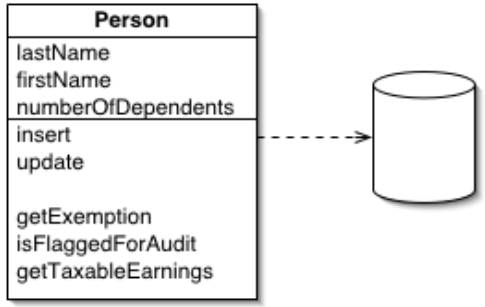
- ORM (Object-Relational Mapping)
  - Bridges the gap between relational databases, which are designed around mathematical Set Theory and Object-Oriented programming languages that deal with objects and their behavior.
  - Greatly simplifies writing code for accessing the database
- In Rails, the Model (usually) uses some ORM framework

Design pattern

Active Record

An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

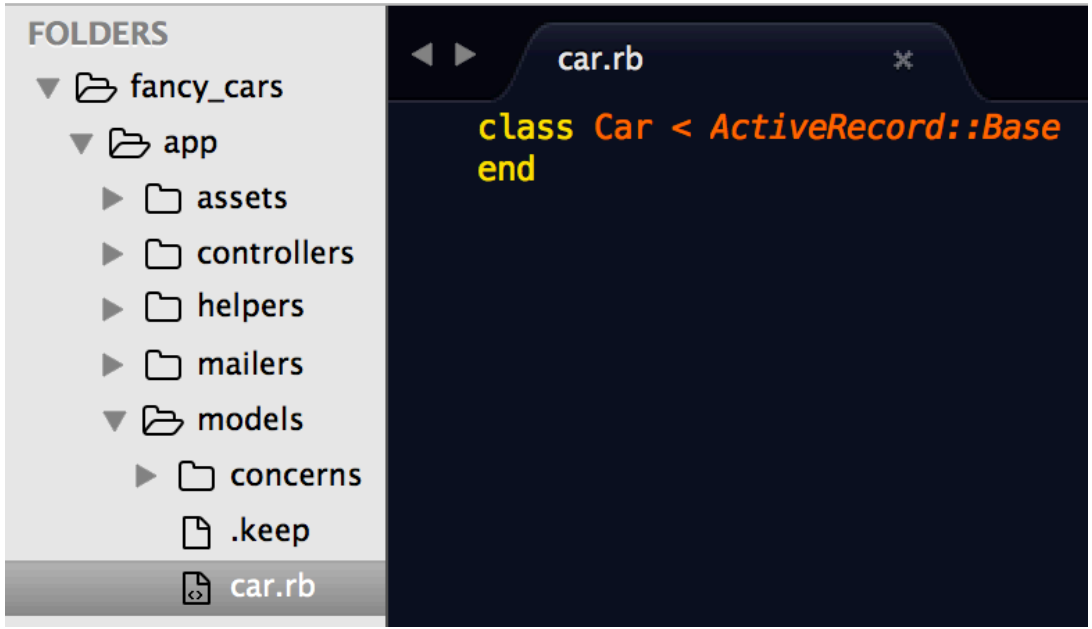
For a full description see P of EAA page **160**



An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.

Active Record

- ***ActiveRecord*** is also the name of Rails' default ORM



Where is all the code?

Metaprogramming + Conventions

ActiveRecord

- Three Prerequisites:

1. ActiveRecord has to know how to find your database (when Rails is loaded, this info is read from `config/database.yml` file)
2. (*Convention*) There is a table with a plural name that corresponds to `ActiveRecord::Base` subclass with a singular name
3. (*Convention*) Expects the table to have a primary key named `id`

Rails Console: rails c

- “IRB on steroids” with your Rails App loaded

```
~/fancy_cars$ rails c
Loading development environment (Rails 4.2.3)
irb(main):001:0> Car.column_names
=> ["id", "company", "color", "year", "created_at", "updated_at", "price"]
irb(main):002:0> Car.primary_key
=> "id"
irb(main):003:0> exit
```

Class methods deal with the table as a whole, while instance methods deal with a particular row of the table…

Model and Migration

- We saw the scaffold generator and the migration generator, but it turns out model has its own generator as well, which could also generate a migration

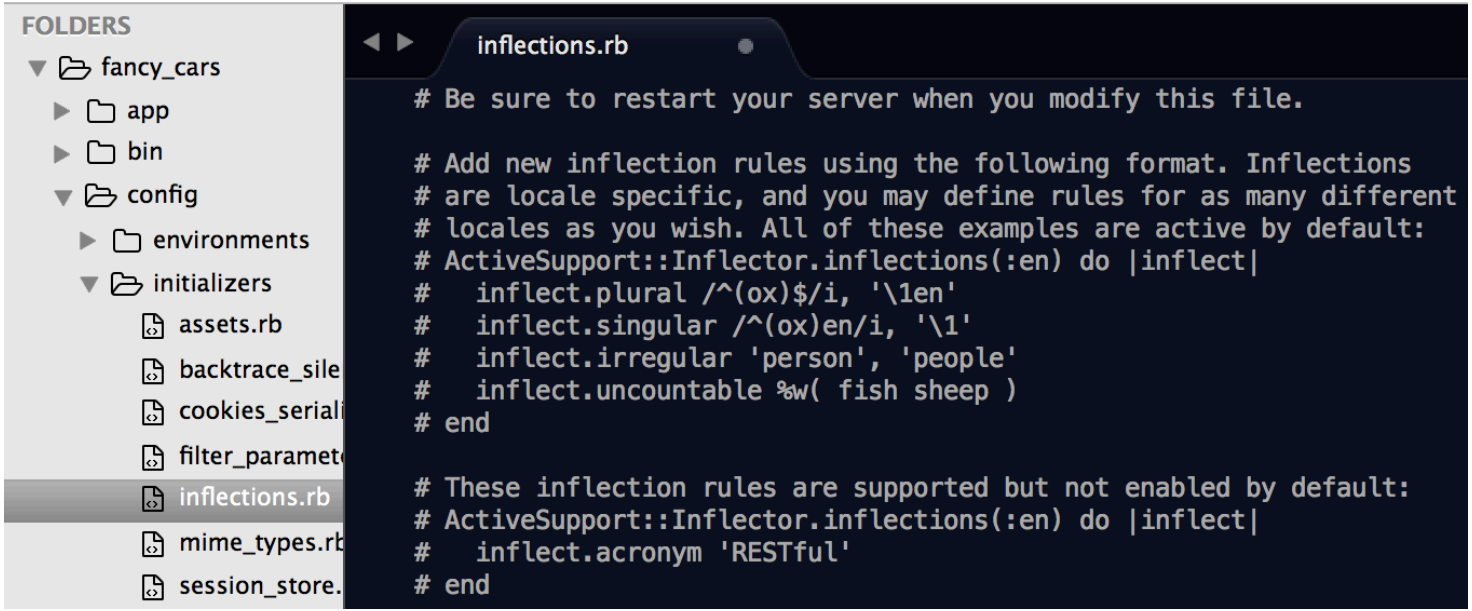
```
~/fancy_cars$ rails g model person first_name last_name
      invoke  active_record
      create    db/migrate/20150907200327_create_people.rb
      create    app/models/person.rb
      invoke    test_unit
      create      test/models/person_test.rb
      create      test/fixtures/people.yml
```

Model and Migration

```
~/fancy_cars$ rake db:migrate
== 20150907200327 CreatePeople: migrating ==========
-- create_table(:people)
   -> 0.0006s
== 20150907200327 CreatePeople: migrated (0.0006s) =
```

Smart enough to know that the table name is **people** not persons

People Not Persons



Reloading Rails Console

- Don’t have to kill rails console after a new migration – just call `reload`!

```
irb(main):011:0> begin
irb(main):012:1* Person.column_names
irb(main):013:1> rescue Exception => e
irb(main):014:1> print e.message
irb(main):015:1> end
Could not find table 'people'=> nil
irb(main):016:0> reload!
Reloading...
=> true
irb(main):017:0> Person.column_names
=> ["id", "first_name", "last_name", "created_at", "updated_at"]
irb(main):018:0>
```

Summary

- ✧ Active Record conventions:
  - Class name is singular
  - DB table name is plural
  - Need to have an `id` primary key

What's Next?

- ✧ Active Record CRUD

In this lecture, we will discuss…

- ✧ Active Record **CRUD**
  - **C**reate
  - **R**etrieve

Create (***C***RUD)

- Three ways to create a record in the database
  1. Use an empty constructor and (ghost) attributes to set the values and then call `save`
  2. Pass a hash of attributes into the constructor and then call `save`
  3. Use `create` method with a hash to create an object and save it to the database in one step

```
~/fancy_cars$ rails c
Loading development environment (Rails 4.2.3)
irb(main):001:0> Person.column_names
=> ["id", "first_name", "last_name", "created_at", "updated_at"]
irb(main):002:0> p1 = Person.new; p1.first_name = "Joe"; p1.last_name = "Smith"
=> "Smith"
irb(main):003:0> p1.save
   (0.2ms)  begin transaction
  SQL (1.1ms)  INSERT INTO "people" ("first_name", "last_name", "created_at", "updated_at") VALUES (?, ?, ?, ?)  [["first_name"
, "Joe"], ["last_name", "Smith"], ["created_at", "2015-09-08 02:08:10.357211"], ["updated_at", "2015-09-08 02:08:10.357211"]]
   (0.6ms)  commit transaction
=> true
irb(main):004:0> p2 = Person.new( first_name: "John", last_name: "Doe"); p2.save
   (0.1ms)  begin transaction
  SQL (0.3ms)  INSERT INTO "people" ("first_name", "last_name", "created_at", "updated_at") VALUES (?, ?, ?, ?)  [["first_name"
, "John"], ["last_name", "Doe"], ["created_at", "2015-09-08 02:09:11.329095"], ["updated_at", "2015-09-08 02:09:11.329095"]]
   (1.6ms)  commit transaction
=> true
irb(main):005:0> p3 = Person.create(first_name: "Jane", last_name: "Doe")
   (0.1ms)  begin transaction
  SQL (0.3ms)  INSERT INTO "people" ("first_name", "last_name", "created_at", "updated_at") VALUES (?, ?, ?, ?)  [["first_name"
, "Jane"], ["last_name", "Doe"], ["created_at", "2015-09-08 02:11:18.824233"], ["updated_at", "2015-09-08 02:11:18.824233"]]
   (1.4ms)  commit transaction
=> #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:11:18">
```

Retrieve / Read (C*R*UD)

- `find(id)` or `find(id1, id2)`
  - Throws a `RecordNotFound` exception if not found
- `first, last, take, all`
  - Return the results you expect or `nil` if nothing is found
- `order(:column)` or `order(column: :desc)`
  - Allows ordering of the results. Ascending or descending
- `pluck`
  - Allows to narrow down which fields are coming back
  - Need to call at the end!

```
irb(main):001:0> Person.all.order(first_name: :desc)
  Person Load (1.5ms)  SELECT "people".* FROM "people"  ORDER BY "people"."first_name" DESC
=> #<ActiveRecord::Relation [#<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_a
t: "2015-09-08 02:09:11">, #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at
: "2015-09-08 02:08:10">, #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at:
"2015-09-08 02:11:18">]>
irb(main):002:0> Person.all.order(first_name: :desc).to_a
  Person Load (0.2ms)  SELECT "people".* FROM "people"  ORDER BY "people"."first_name" DESC
=> [#<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_at: "2015-09-08 02:09:11">
, #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10">,
 #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:11:18">]
irb(main):003:0> Person.first
  Person Load (0.2ms)  SELECT  "people".* FROM "people"  ORDER BY "people"."id" ASC LIMIT 1
=> #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10">
irb(main):004:0> Person.all.first
  Person Load (0.2ms)  SELECT  "people".* FROM "people"  ORDER BY "people"."id" ASC LIMIT 1
=> #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10">
irb(main):005:0> Person.all[0]
  Person Load (0.1ms)  SELECT "people".* FROM "people"
=> #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10">
```

Take and Pluck

```
irb(main):001:0> Person.take
  Person Load (1.0ms)  SELECT  "people".* FROM "people" LIMIT 1
=> #<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10">
irb(main):002:0> Person.take 2
  Person Load (0.2ms)  SELECT  "people".* FROM "people" LIMIT 2
=> [#<Person id: 1, first_name: "Joe", last_name: "Smith", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 02:08:10"
>, #<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_at: "2015-09-08 02:09:11">]
irb(main):003:0> Person.all.map { |person| person.first_name }
  Person Load (0.2ms)  SELECT "people".* FROM "people"
=> ["Joe", "John", "Jane"]
irb(main):004:0> Person.pluck(:first_name)
   (0.2ms)  SELECT "people"."first_name" FROM "people"
=> ["Joe", "John", "Jane"]
```

Retrieve / Read (C*R*UD)

- `where(hash)`
  - Enables you to supply conditions for your search
  - Returns `ActiveRecord::Relation` (same as `all`), but you can always narrow it down with `first` or treat it like an Array…

Retrieve / Read (C*R*UD) - where

```
irb(main):001:0> Person.where(last_name: "Doe")
  Person Load (0.1ms)  SELECT "people".* FROM "people" WHERE "people"."last_name" = ?  [["last_name", "Doe"]]
=> #<ActiveRecord::Relation [#<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_a
t: "2015-09-08 02:09:11">, #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at:
 "2015-09-08 02:11:18">]>
irb(main):002:0> Person.where(last_name: "Doe").first
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."last_name" = ?  ORDER BY "people"."id" ASC LIMIT 1  [["
last_name", "Doe"]]
=> #<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_at: "2015-09-08 02:09:11">
irb(main):003:0> Person.where(last_name: "Doe")[0]
  Person Load (0.2ms)  SELECT "people".* FROM "people" WHERE "people"."last_name" = ?  [["last_name", "Doe"]]
=> #<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_at: "2015-09-08 02:09:11">
irb(main):004:0> Person.where(last_name: "Doe").pluck(:first_name)
   (0.2ms)  SELECT "people"."first_name" FROM "people" WHERE "people"."last_name" = ?  [["last_name", "Doe"]]
=> ["John", "Jane"]
```

Find_by

- `find_by(conditions_hash)`
  - Same as `where`, but returns a single result or nil if a record with the specified conditions is not found
- `find_by!(conditions_hash)`
  - Same as `find_by`, but throws an exception if cannot find the result

Find_by and Find_by!

```
irb(main):001:0> Person.find_by(last_name: "Doe")
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."last_name" = ? LIMIT 1  [["last_name", "Doe"]]
=> #<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_at: "2015-09-08 02:09:11">
irb(main):002:0> Person.where(last_name: "Doe")
  Person Load (0.3ms)  SELECT "people".* FROM "people" WHERE "people"."last_name" = ?  [["last_name", "Doe"]]
=> #<ActiveRecord::Relation [#<Person id: 2, first_name: "John", last_name: "Doe", created_at: "2015-09-08 02:09:11", updated_a
t: "2015-09-08 02:09:11">, #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at:
 "2015-09-08 02:11:18">]>
irb(main):003:0> Person.find_by(last_name: "Nosuchdude")
  Person Load (0.4ms)  SELECT  "people".* FROM "people" WHERE "people"."last_name" = ? LIMIT 1  [["last_name", "Nosuchdude"]]
=> nil
irb(main):004:0> Person.find_by!(last_name: "Incognito")
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."last_name" = ? LIMIT 1  [["last_name", "Incognito"]]
ActiveRecord::RecordNotFound: Couldn't find Person
```

limit / offset

- `limit(n)`
  - Enables you to limit how many records come back
- `offset(n)`
  - Don’t start from the beginning; skip a few
- You can combine these two to “page” through large collections of records in your database

limit / offset

```
irb(main):001:0> Person.count
   (0.1ms)  SELECT COUNT(*) FROM "people"
=> 3
irb(main):002:0> Person.all.map { |person| "#{person.first_name} #{person.last_name}" }
  Person Load (0.3ms)  SELECT "people".* FROM "people"
=> ["Joe Smithson", "John Doe", "Jane Smithie"]
irb(main):003:0> Person.offset(1).limit(1).map { |person| "#{person.first_name} #{person.last_name}" }
  Person Load (0.2ms)  SELECT  "people".* FROM "people" LIMIT 1 OFFSET 1
=> ["John Doe"]
irb(main):004:0> Person.offset(1).limit(1).all.map { |person| "#{person.first_name} #{person.last_name}" }
  Person Load (0.1ms)  SELECT  "people".* FROM "people" LIMIT 1 OFFSET 1
=> ["John Doe"]
```

Summary

- ActiveRecord is very intuitive when it comes to interacting with a database
- Always keep in mind if you are getting back a single result (find) or an `ActiveRecord::Relation` (where)

What's Next?

- Active Record CRUD continued

In this lecture, we will discuss…

- ✧ Active Record **CRUD**
  - **U**pdate
  - **D**elete

Update (CR*U*D)

- Two ways to update a record in the database:
  1. Retrieve a record, modify the values and then call `save`
  2. Retrieve a record and then call `update` method passing in a hash of attributes with new values

- There is also `update_all` for batch updates
  - You can chain this to the end of `where`

Update (CR*U*D)

```
irb(main):001:0> jane = Person.find_by first_name: "Jane"
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."first_name" = ? LIMIT 1  [["first_name", "Jane"]]
=> #<Person id: 3, first_name: "Jane", last_name: "Doe", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:11:18">
irb(main):002:0> jane.last_name = "Smithie"
=> "Smithie"
irb(main):003:0> jane.save
   (0.2ms)  begin transaction
  SQL (0.4ms)  UPDATE "people" SET "last_name" = ?, "updated_at" = ? WHERE "people"."id" = ?  [["last_name", "Smithie"], ["upda
ted_at", "2015-09-08 02:59:20.775235"], ["id", 3]]
   (1.4ms)  commit transaction
=> true
irb(main):004:0> jane = Person.find(3)
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."id" = ? LIMIT 1  [["id", 3]]
=> #<Person id: 3, first_name: "Jane", last_name: "Smithie", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:59:2
0">
irb(main):005:0> Person.find_by(last_name: "Smith").update(last_name: "Smithson")
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."last_name" = ? LIMIT 1  [["last_name", "Smith"]]
   (0.1ms)  begin transaction
  SQL (0.3ms)  UPDATE "people" SET "last_name" = ?, "updated_at" = ? WHERE "people"."id" = ?  [["last_name", "Smithson"], ["upd
ated_at", "2015-09-08 03:00:33.037717"], ["id", 1]]
   (1.4ms)  commit transaction
=> true
```

Delete (CRU*D*)

- `destroy(id)` or `destroy`
  - Removes a particular instance from the DB
  - Instantiates an object first and performs callbacks before removing
  - See http://guides.rubyonrails.org/active_record_callbacks.html
- `delete(id)`
  - Removes the row from DB
- There is also a `delete_all` CAREFUL!

Delete / Destroy

```
irb(main):001:0> Person.count
   (0.1ms)  SELECT COUNT(*) FROM "people"
=> 3
irb(main):002:0> jane = Person.find_by first_name: "Jane"
  Person Load (0.2ms)  SELECT  "people".* FROM "people" WHERE "people"."first_name" = ? LIMIT 1  [["first_name", "Jane"]]
=> #<Person id: 3, first_name: "Jane", last_name: "Smithie", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:59:20">
irb(main):003:0> jane.destroy
   (0.2ms)  begin transaction
  SQL (0.5ms)  DELETE FROM "people" WHERE "people"."id" = ?  [["id", 3]]
   (1.5ms)  commit transaction
=> #<Person id: 3, first_name: "Jane", last_name: "Smithie", created_at: "2015-09-08 02:11:18", updated_at: "2015-09-08 02:59:20">
irb(main):004:0> joe = Person.find_by first_name: "Joe"
  Person Load (0.1ms)  SELECT  "people".* FROM "people" WHERE "people"."first_name" = ? LIMIT 1  [["first_name", "Joe"]]
=> #<Person id: 1, first_name: "Joe", last_name: "Smithson", created_at: "2015-09-08 02:08:10", updated_at: "2015-09-08 03:00:33">
irb(main):005:0> Person.delete(joe.id)
  SQL (2.8ms)  DELETE FROM "people" WHERE "people"."id" = ?  [["id", 1]]
=> 1
irb(main):006:0> Person.count
   (0.2ms)  SELECT COUNT(*) FROM "people"
=> 1
```

Summary

- Update and Delete are both simple to use
- Delete has a
  - 1. “go straight to DB version” (`delete`)
  - 2. Instantiate Ruby object and let it interact with DB when it’s ready version (`destroy`)