Basic Rails Models and Database Access

Professor Larry Heimann
Application Design & Development
Information Systems Program

With additional slides from Prof. Houda Bouamor, CMU-Q

Why Ruby on Rails?

“Ruby on Rails is astounding. Using it is like watching a kung-fu movie, where a dozen bad-ass frameworks prepare to beat up on the little newcomer only to be handed their asses in a variety of imaginative ways.”

— Nathan Torkington
O'Reilly Program Chair for OSCON

Rails Basic Commands

- The ones we already know:
  - `rails new <app_name>`
  - `rails generate   #or rails g`
  - `rails server  #or rails s`
  - `rails server -p <port number> #e.g., rails server -p 3030`
  - `rails routes`
  - `rails db:migrate`
  - `rails console  # or rails c`
  - `rails –help #check all the rails commands available for you`
- The ones we will learn about later this semester
  - `rails test`
  - `rails db:rollback`

Model basics

- ActiveRecord does the heavy lifting
- Basic relationships and scopes
- Validations to ensure data integrity
- Other methods added as needed

Key Idea:
Models hold all the data
& business logic

Goals of Active Record

- Make working with databases easier
- Reduce repetition in code
- Cut down on configuration needed to make applications work

Object-Relational Mapping (ORM)

- **Database access before ORM:** Using special APIs
  - Prepare a SQL statement
  - Issue a query
  - Get record from the result
  - Extract field(s) from the record

- **Object Relational Mapping (ORM):** simplifies the use of databases in applications.
  - Provides an interface and binding between the tables in a relational database
  - Uses objects to hold database records
  - Manages the movement of information between objects and the back-end database.
  - Manage relationships between tables (joins): using linked data structures.

ActiveRecord: The Rails ORM

- **ActiveRecord** is the ORM layer that is supplied with Rails. It is the part of Rails that implements your application's model.

- **ActiveRecord**: "An **object** that wraps a **row** in a database table or view, encapsulates the database **access**, and adds **domain logic** on that data."
  Martin Fowler, Patterns of Enterprise Application Architecture (2002), Page 160

- Each subclass of **ActiveRecord::Base** (i.e., Proverb class), wraps a separate database table (i.e., proverbs table).

- ActiveRecord models automatically provide getter and setter methods related to each field in the model. Additional getter and setter methods can be added later if virtual attributes are needed.

Connecting to a Database

- **Key 1:** database.yml
  - Located in the config/ directory
  - Contains connection information for 3+ levels
    - development
    - test
    - production
    - others possible, such as staging

- **Key 2:** Gemfile provides adapter
  - Typically use sqlite3 in development & test
  - Typically use Postgres or MySQL in production

Building the Database with Migrations

```
class CreatePets < ActiveRecord::Migration
  def change
    create_table :pets do |t|
      t.integer :animal_id
      t.integer :owner_id
      t.string :name
      t.boolean :female
      t.date :date_of_birth
      t.boolean :active

      t.timestamps
    end
  end
end
```

Rails migrations

- Migrations are stored as files in the **db/migrate** directory, one for each migration class.
- The name of the file is of the form `YYYYMMDDHHMMSS_create_proverbs.rb`
    - Rails uses the **timestamp** to determine which migration should be run and in what order.

- Migrations are used to:
    - Create tables
    - Change tables: add column, rename column, remove column, etc.
    - Drop tables

Primary Key and Active Record

- Active Record gives you a way of overriding the default name of the primary key for a table.
- For example, we may be working with an existing table that uses the ISBN as the primary key for the books table.

```
class Book < ApplicationRecord

  self.primary_key = "isbn"

end
```

- Problem is that a lot of Rails shortcuts can be short-circuited by this
- Read more about this: *Agile Web Development with Rails 5.1, chapter 20, Locating and Traversing Records:* https://learning.oreilly.com/library/view/agile-web-development/9781680502985/f_0107.xhtml

Migration Data Types

	mysql	openbase	oracle	postgresql	sqlite	sqlserver
:binary	blob	object	blob	bytea	blob	image
:boolean	tinyint(1)	boolean	number(1)	boolean	boolean	bit
:date	date	date	date	date	date	datetime
:datetime	datetime	datetime	date	timestamp	datetime	datetime
:decimal	decimal	decimal	decimal	decimal	decimal	decimal
:float	float	float	number	float	float	float(8)
:integer	int(11)	integer	number(38)	integer	integer	int
:string	varchar(255)	char(4096)	varchar2(255)	(note 1)	varchar(255)	varchar(255)
:text	text	text	clob	text	text	text
:time	time	time	date	time	datetime	datetime
:timestamp	datetime	timestamp	date	timestamp	datetime	datetime

ActiveRecord: Fundamentals

- Table names are plural and class names are singular

Class Name	Table Name	Class Name	Table Name
Order	orders	LineItem	line_items
TaxAgency	tax_agencies	Person	people
Batch	batches	Datum	data
Diagnosis	diagnoses	Quantity	quantities

- If class contains multiple capital words, table has underscore between words : i.e, TaxAgency -> tax_agencies

Plurals vs. Singular in Rails

OwnersController

owners *table*

Pet *model*

has_many :pets

@owner.pets

belongs_to :owner

@pet.owner

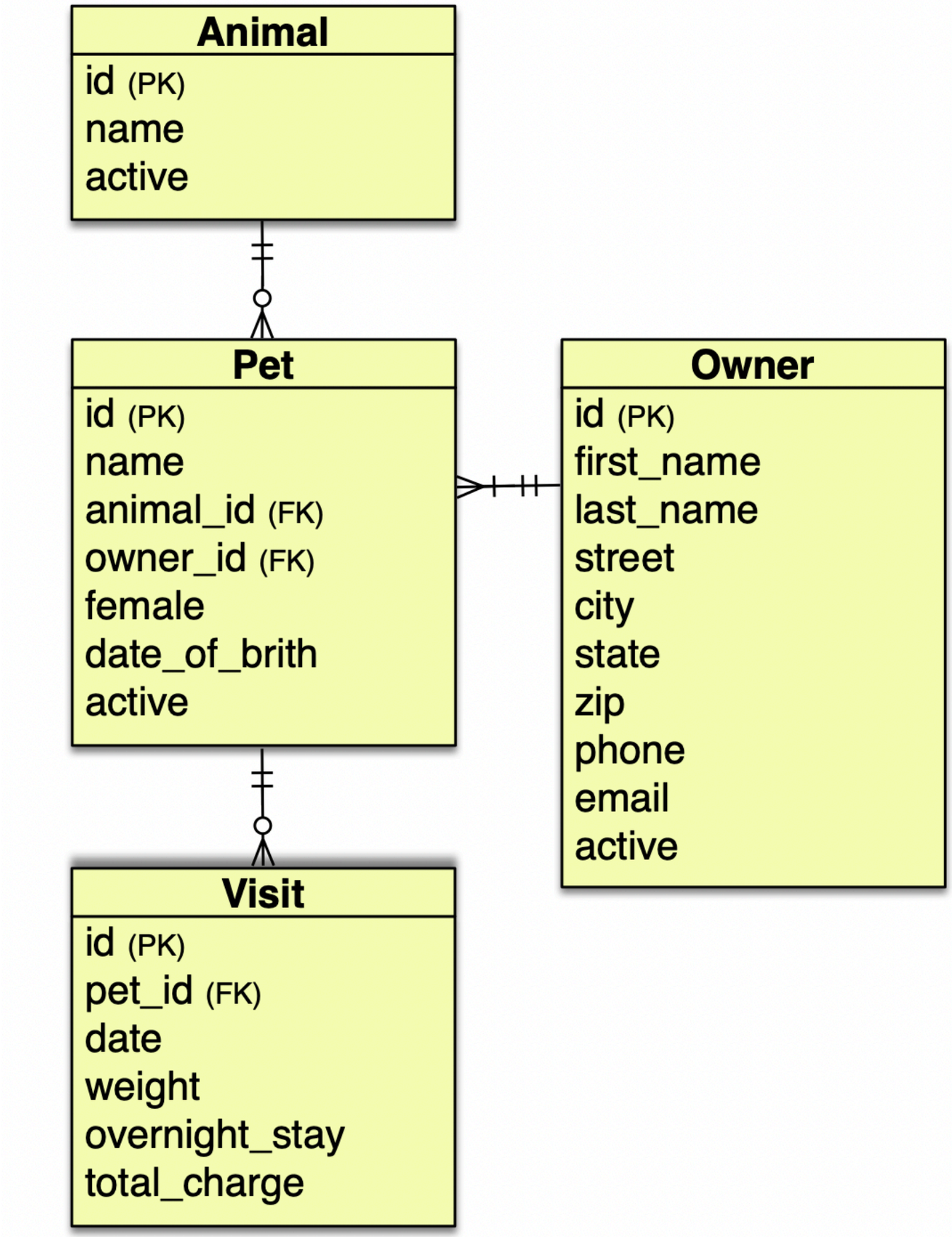
Associations in Rails

- In Rails, a relationship or an association is a **connection** between two Active Record **models**

- Why Relationships?
  - Because they make common operations simpler and easier in your code.

- Relationships allow us to add foreign key information in Rails model

A slice of the PATS system



Relationships

(see examples in PATS)

ActiveRecord creates SQL

(examples given in class)

Scenario: Owner and Pet models

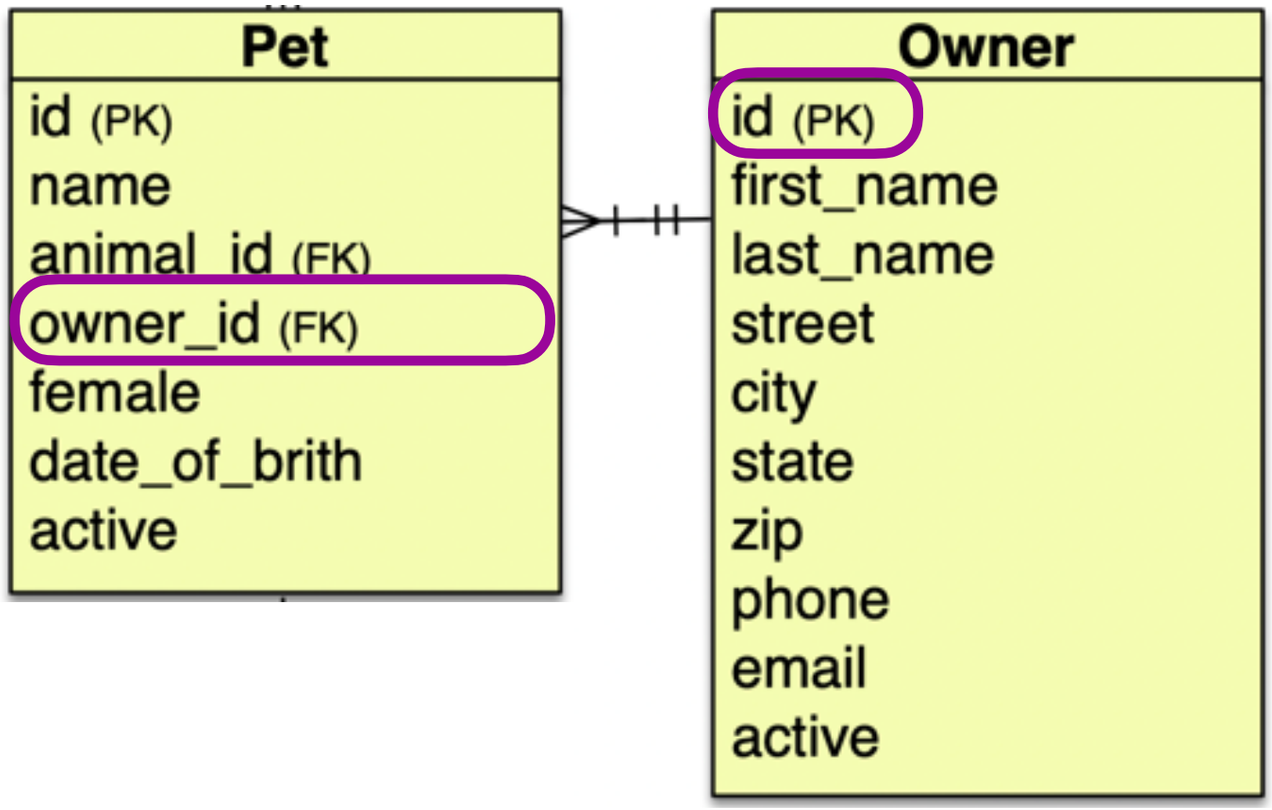
- For example, our Rails application includes a model for owners and a model for pets.

- Each owner can have many pets. Without associations, the model declarations looks like this:

```
class Owner < ApplicationRecord
end

class Pet < ApplicationRecord
end
```

Scenario: Owner and Pet models

- To add a new pet for an existing owner, we'd need to include the owner_id as a **foreign key.**



- When we'd like to delete owner having id=1, we should ensure that all his pets get deleted as well.

In Rails

- With Active Record associations, we can streamline these - and other - operations by declaratively telling Rails that there is a **connection between the two models.**

- Active Record supports the three common types of relationship between tables:
    1. one-to-one
    2. one-to-many
    3. many-to-many

- These relationships are indicated by adding declarations to **models**: **has_one**, **has_many**, **belongs_to**, and the wonderfully named **has_and_belongs_to_many.**

Types of Associations

- Rails supports six types of associations:

  1.belongs_to

  2.has_many

  3.has_many :through

  4.has_one

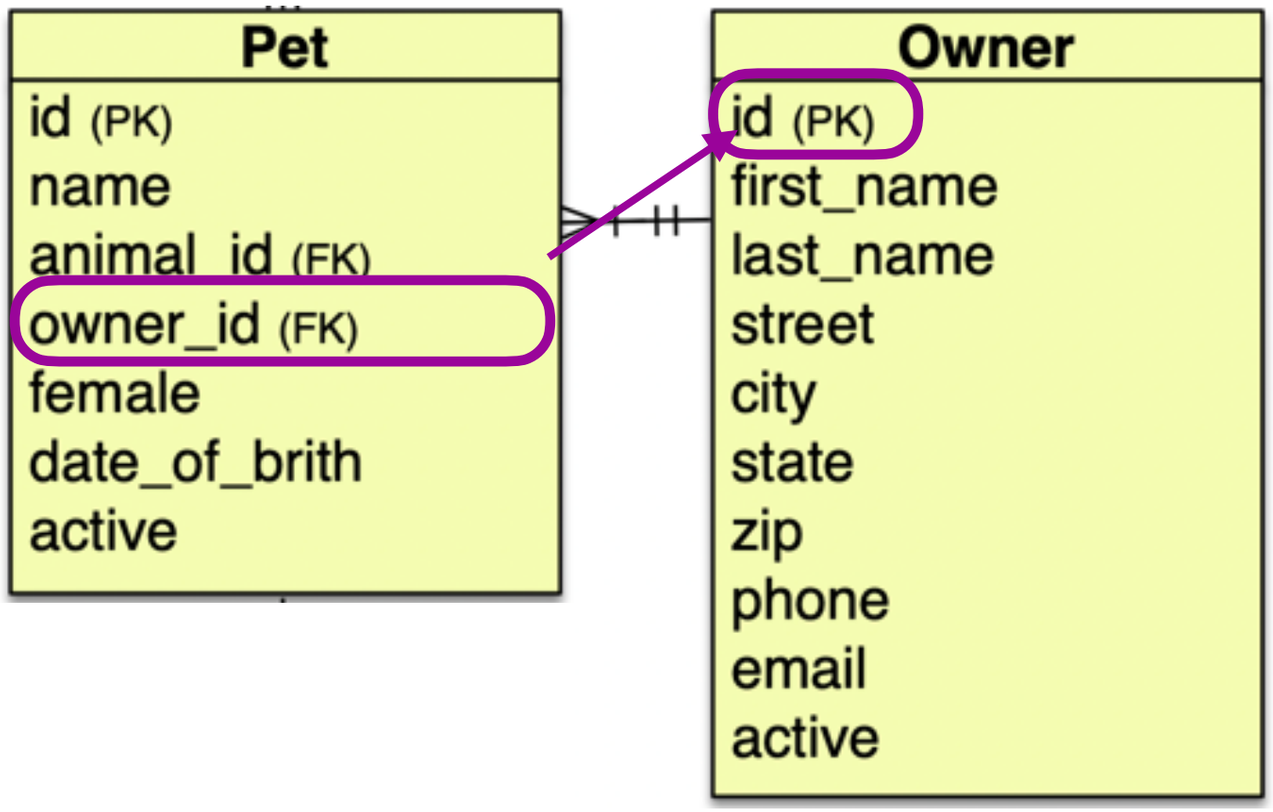
  5.has_one :through

  6.has_and_belongs_to_many

The belongs_to Association

- A `belongs_to` association sets up a one-to-one connection with another model, such that each instance of the declaring model "belongs to" one instance of the other model.

- For example, and each pet can be assigned to exactly one owner.

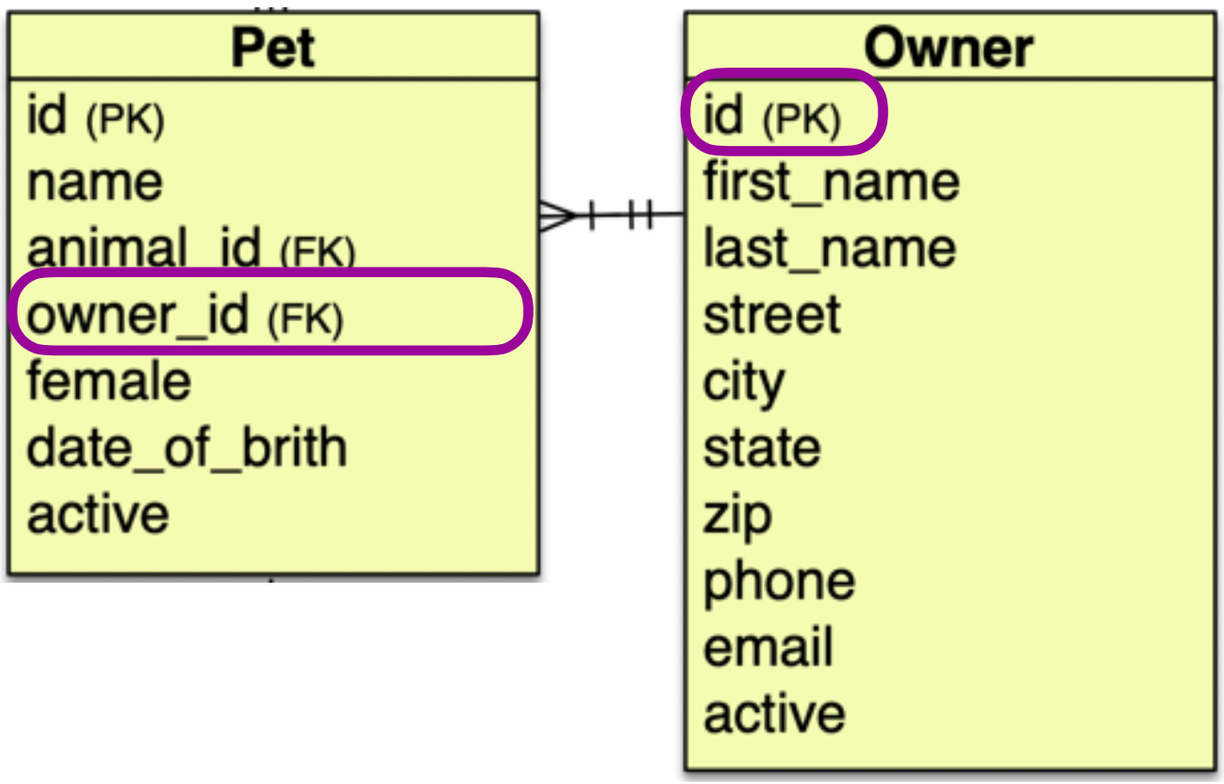



```
class Pet < ApplicationRecord

 belongs_to :owner

end
```

! belongs_to associations must use the singular term (owner and not owners!).

The has_many Association

- A **has_many** association indicates a **one-to-many** connection with another model
- You'll often find this association on the "other side" of a belongs_to association
- This association indicates that each instance of the model has zero or more instances of another model.
- For example, and each owner can be have assigned to it many pets.



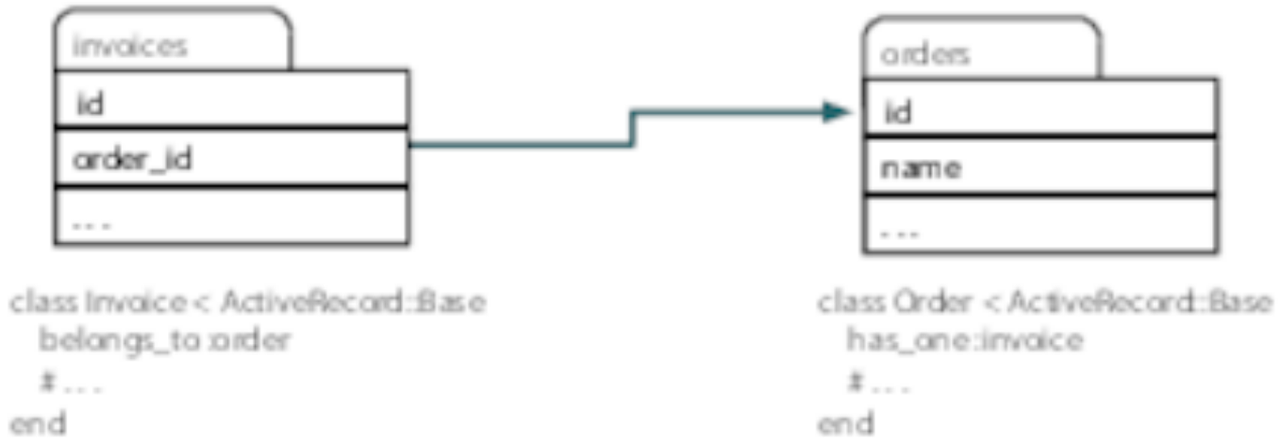

```
class Owner < ApplicationRecord

 has_many :pets

end
```

! has_many associations must use the plural term (pets and not pet!).

The has_one Association

- The **has_one** association creates a **one-to-one** match with another model.

- In database terms, this association says that the other class contains the foreign key. If this class contains the foreign key, then you should use **belongs_to** instead.



- There’s an important rule illustrated here: the model for the table that contains the foreign key always has the **belongs_to** declaration.

The has_and_belongs_to_many Association

- An employee can belong to several projects
- And each project may have several employees.
- This is an example of a many-to-many relationships.

- In Rails we can express this by adding the has_and_belongs_to_many declaration to both models.

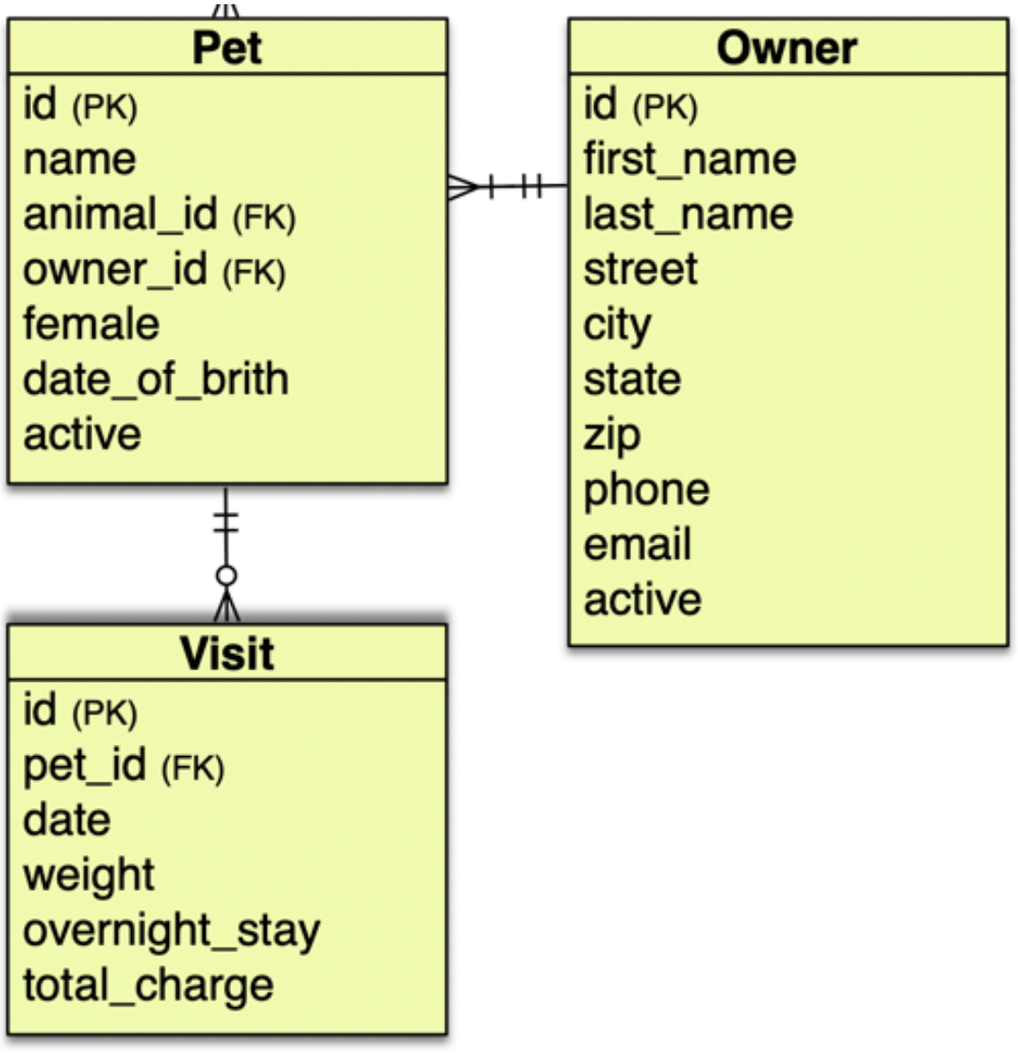
The has_and_belongs_to_many Association


```
class Employee < ApplicationRecord

  has_and_belongs_to_many :projects

  # …

end
```

```
class Project < ApplicationRecord

has_and_belongs_to_many :employees

 # …

end
```

The has_many :through Association

- A **has_many :through** association is often used to set up a **many-to-many** connection with another model.
- This association indicates that the declaring model can be matched with zero or more instances of another model by proceeding through a third model.




```
class Owner < ApplicationRecord

    has_many :pets
    has_many :visits, through: :pets

end
```

The has_one through: Association

- A **has_one** :through association sets up a **one-to-one** connection with another model.
- This association indicates that the declaring model can be matched with one instance of another model by proceeding **through a third model.**

```
class Visit < ApplicationRecord
  belongs_to :pet
  has_one :animal, through: :pet
  has_one :owner, through: :pet
end
```

Choosing Between belongs_to and has_one

- If you want to set up a one-to-one relationship between two models, you'll need to add **belongs_to** to one, and **has_one** to the other. How do you know which is which?

- The distinction is in where you place the foreign key (it goes on the table for the class declaring the belongs_to association), but you should give some thought to the actual meaning of the data as well.

- The has_one relationship says that one of something is yours - that is, that something points back to you.

- For example, it makes more sense to say that a supplier owns an account than that an account owns a supplier.

The Farm, Chicken and Egg example

```
class Farm < ApplicationRecord
  has_many :chickens
  has_many :eggs, through: :chickens
end
```

```
class Chicken < ApplicationRecord
  belongs_to :farm
  has_many :eggs
end
```

```
class Egg < ApplicationRecord
  belongs_to :chicken
  has_one :farm, through: :chicken
end
```

More on Associations

https://guides.rubyonrails.org/association_basics.html

Read more about ORM and ActiveRecord

- Agile Web Development with Rails 5.1, *Chapters 3, “Rails Model Support”*
- Agile Web Development with Rails 5.1, *Chapters 20, “Active Record”*

Read more about Migrations

- Agile Web Development with Rails 5.1, *Chapters 23, “Migrations”*