CHAPTER 3

# Rails Models

# Model

The Model maintains the relationship between objects and the database and handles validation, association, transactions etc.

This subsystem is implemented in the ActiveRecord library which provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records.

Ruby method names are automatically generated from the field names of database tables, and so on.

When a model is generated by rails a migration file is created in db/migrate, this is named using the current date/time as part of the file name and a class file is created in app/models directory that matches the model created, an example of these are shown below.

Generate the model and define the fields and data types.

rails g model Employee name:text phone:text department\_id:integer

This would produce:

db/migrate/20181031090813\_create\_employees.rb

class CreateEmployees < ActiveRecord::Migration[5.0]

def change

create\_table :employees do |t|

t.text :name

t.text :phone

t.integer :department\_id

t.timestamps

end

end

end

Note: as the migration file is named based on the current date/time of the execution of the generate command your migration file will have a slightly different named to the example shown above.

ActiveRecord is one of the most flexible elements of the Rails environment.

The functionality required is initiated by calls to objects in the controller, see later topic, and values are returned to the controller for further processing within the application, typically passing the data to a 'view'.

To produce the underlying database and tables we need to run the rake command with the db sub commands

**rake db:create**

Created database 'db/development.sqlite3'

Created database 'db/test.sqlite3'

**rake db:migrate**

== 20181031090813 CreateEmployees: migrating ==================================

-- create\_table(:employees)

-> 0.0013s

== 20181031090813 CreateEmployees: migrated (0.0013s) =========================

## Active record data types

The active record supports several data types, defined in the migration file either by the rails generator or by manual addition by the developer.

The following data types are currently supported by rails 6.

:primary\_key

:string

:text

:integer

:bigint

:float

:decimal

:numeric

:datetime

:time

:date

:binary

:boolean

## Model Associations

Associations are implemented using simple macro-type definitions.

By declaring that one model **belongs\_to** another, you instruct Rails to maintain the **Primary Key–Foreign Key** information between instances of the two models.

Rails supports six types of associations:

belongs\_to

has\_one

has\_many

has\_many :through

has\_one :through

has\_and\_belongs\_to\_many

### belongs\_to

A belongs\_to association, one of the most commonly used, 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, if your application includes departments and employees, and each employee can be assigned to exactly one department.

|  |
| --- |
| **department** |
| **id** |
| **Name** |
| **location** |

|  |
| --- |
| **employee** |
| **id** |
| **name** |
| **phone** |
| **department** |

You'd declare the order model this way:

In app/models/employee.rb

class Employee < ApplicationRecord

belongs\_to :department

end

or as part of the model generation, the preferred method:

rails g model employee name:text phone:text department:reference

This produces a migration file for the employee table as shown below:

class CreateEmployees < ActiveRecord::Migration[6.0]

def change

create\_table :employees do |t|

t.text :name

t.text :phone

t.references :department, null: false, foreign\_key: true

t.timestamps

end

end

end

When the migration is executed an additional column is added to the database called:

department\_id

This allows us to do the following in app/views/employee/index.html.erb

<% @employees.each do |employee| %>

<tr>

<td><%= employee.name %></td>

<td><%= employee.phone %></td>

<td><%= employee.department.name %></td> **<- The relationship**

<td><%= link\_to 'Show', employee %></td>

<td><%= link\_to 'Edit', edit\_employee\_path(employee) %></td>

<td><%= link\_to 'Destroy', employee, method: :delete, data: { confirm: 'Are you sure?' } %></td>

</tr>

<% end %>

Note: the reference is

employee.department.name

not

employee.department\_id.name

### has\_one

A has\_one association also sets up a one-to-one connection with another model, but with somewhat different semantics (and consequences).

This association indicates that each instance of a model contains or possesses one instance of another model.

For example, if each supplier in your application has only one account, you'd declare the supplier model like this

class Supplier < ActiveRecord::Base

  has\_one :account

end

### 

### has\_many

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, in an application containing customers and orders, the customer model could be declared like this:

class Customer < ActiveRecord::Base

  has\_many :orders

end

### has\_many :through

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.

### has\_one :through

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.

### has\_and\_belongs\_to\_many

A has\_and\_belongs\_to\_many association creates a direct many-to-many connection with another model, with no intervening model.