Active Record Associations

This guide covers the association features of Active Record.

After reading this guide, you will know:

How to declare associations between Active Record models.

How to understand the various types of Active Record associations.

How to use the methods added to your models by creating associations.

Chapters

1 Why Associations?

Why do we need associations between models? Because they make common operations simpler and easier in your code. For example, consider a simple Rails application that includes a model for customers and a model for orders. Each customer can have many orders. Without associations, the model declarations would look like this:

```
class Customer < ActiveRecord::Base
end

class Order < ActiveRecord::Base
end
```

Now, suppose we wanted to add a new order for an existing customer. We'd need to do something like this:

```
@order = Order.create(order_date: Time.now, customer_id:
@customer.id)
```

Or consider deleting a customer, and ensuring that all of its orders get deleted as well:

```
@orders = Order.where(customer_id: @customer.id)
@orders.each do |order|
  order.destroy
end
@customer.destroy
```

With Active Record associations, we can streamline these - and other - operations by declaratively telling Rails that there is a connection between the two models. Here's the revised code for setting up customers and orders:

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :destroy
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

With this change, creating a new order for a particular customer is easier:

```
@order = @customer.orders.create(order_date: Time.now)
```

Deleting a customer and all of its orders is *much* easier:

```
@customer.destroy
```

To learn more about the different types of associations, read the next section of this guide. That's followed by some tips and tricks for working with associations, and then by a complete reference to the methods and options for associations in Rails.

2 The Types of Associations

In Rails, an *association* is a connection between two Active Record models. Associations are implemented using macro-style calls, so that you can declaratively add features to your models. For example, by declaring that one model `belongs_to` another, you instruct Rails to maintain Primary Key-Foreign Key information between instances of the two models, and you also get a number of utility methods added to your model. Rails supports six types of associations:

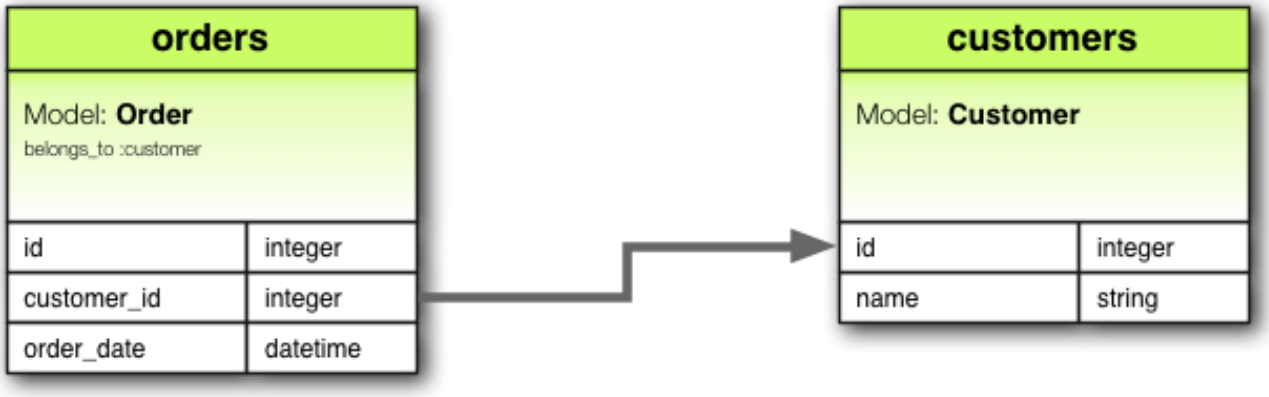
```
belongs_to
has_one
has_many
has_many :through
has_one :through
has_and_belongs_to_many
```

In the remainder of this guide, you'll learn how to declare and use the various forms of associations. But first, a quick introduction to the situations where each association type is appropriate.

2.1 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, if your application includes customers and orders, and each order can be assigned to exactly one customer, you'd declare the order model this way:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```




```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

`belongs_to` associations *must* use the singular term. If you used the pluralized form in the above example for the `customer` association in the `Order` model, you would be told that there was an "uninitialized constant Order::Customers". This is because Rails automatically infers the class name from the association name. If the association name is wrongly pluralized, then the inferred class will be wrongly pluralized too.

The corresponding migration might look like this:

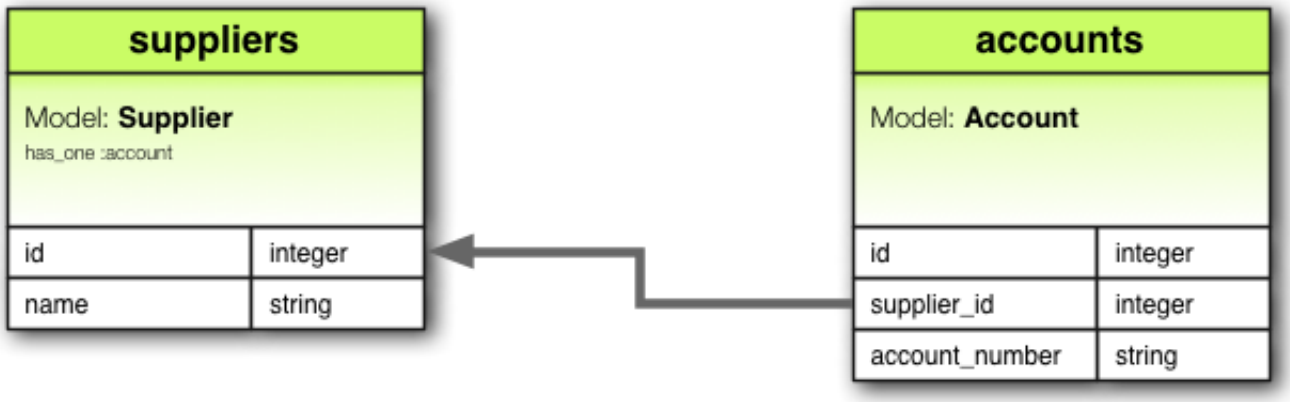
```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :orders do |t|
      t.belongs_to :customer, index: true
```

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

2.2 The `has_one` Association

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
```




```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
```

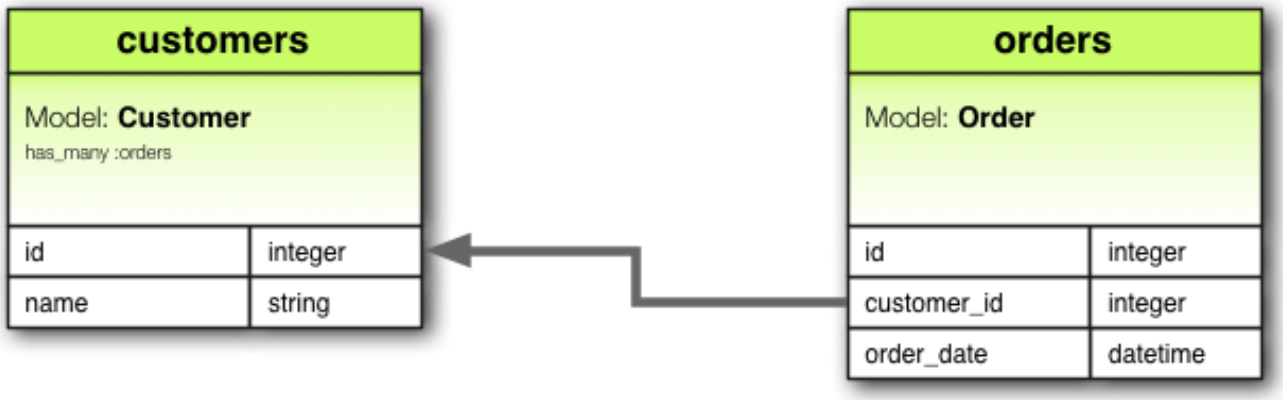
```
      t.string :account_number
      t.timestamps null: false
    end
  end
end
```

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

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

The name of the other model is pluralized when declaring a `has_many` association.




```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

The corresponding migration might look like this:

```
class CreateCustomers < ActiveRecord::Migration
  def change
    create_table :customers do |t|
      t.string :name
      t.timestamps null: false
    end
```

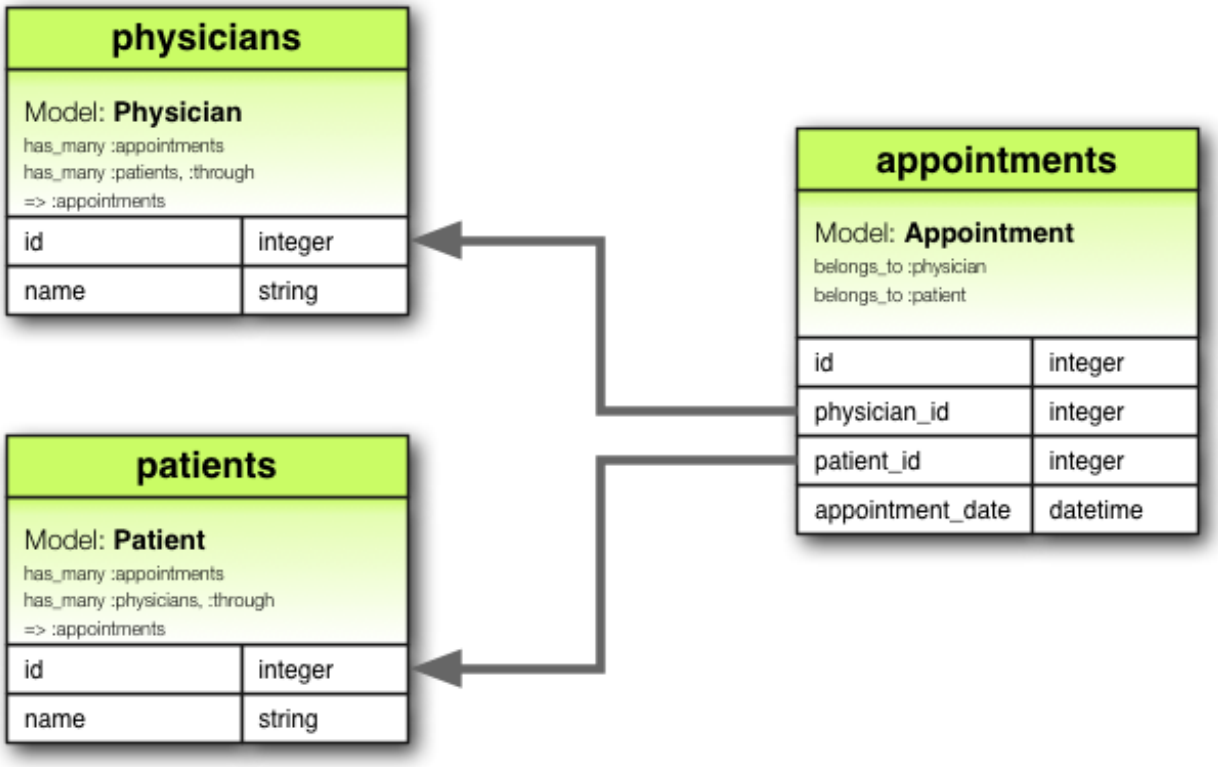
```
    create_table :orders do |t|
      t.belongs_to :customer, index:true
      t.datetime :order_date
      t.timestamps null: false
    end
  end
end
```

2.4 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. For example, consider a medical practice where patients make appointments to see physicians. The relevant association declarations could look like this:

```
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, through: :appointments
end
```

The corresponding migration might look like this:




```ruby
class Physician < ActiveRecord::Base
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ActiveRecord::Base
  belongs_to :physician
  belongs_to :patient
end

class Patient < ActiveRecord::Base
  has_many :appointments
  has_many :physicians, :through => :appointments
end
```

```ruby
class CreateAppointments < ActiveRecord::Migration
  def change
    create_table :physicians do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :patients do |t|
```

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

    create_table :appointments do |t|
      t.belongs_to :physician, index: true
      t.belongs_to :patient, index: true
      t.datetime :appointment_date
      t.timestamps null: false
    end
  end
end
```

The collection of join models can be managed via the API. For example, if you assign

```
physician.patients = patients
```

new join models are created for newly associated objects, and if some are gone their rows are deleted.

Automatic deletion of join models is direct, no destroy callbacks are triggered.

The `has_many :through` association is also useful for setting up "shortcuts" through nested `has_many` associations. For example, if a document has many sections, and a section has many paragraphs, you may sometimes want to get a simple collection of all paragraphs in the document. You could set that up this way:

```
class Document < ActiveRecord::Base
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ActiveRecord::Base
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ActiveRecord::Base
  belongs_to :section
end
```

With `through: :sections` specified, Rails will now understand:

```
@document.paragraphs
```

2.5 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. For example, if each supplier has one account, and each account is associated with one account history, then the supplier model could look like this:

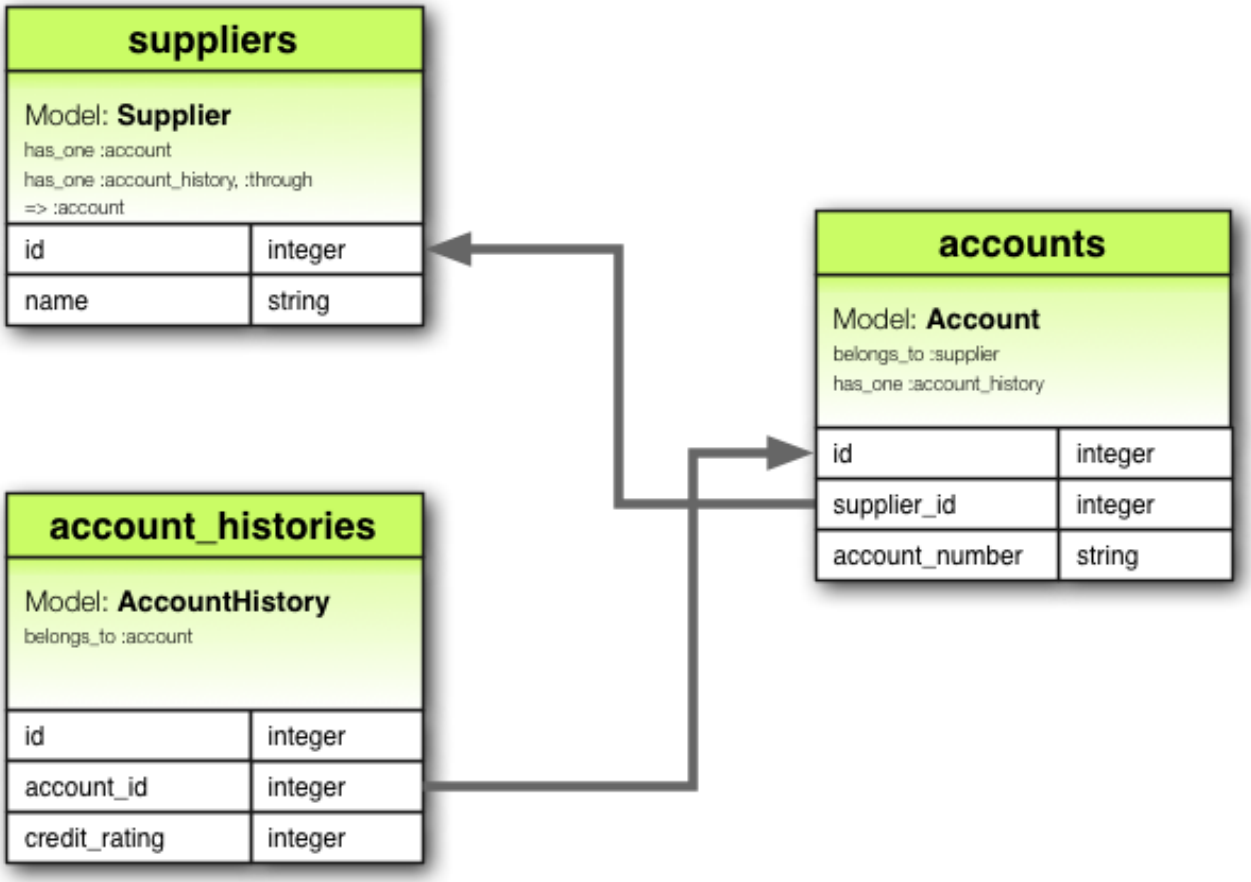
```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, through: :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

The corresponding migration might look like this:

```
class CreateAccountHistories < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.belongs_to :supplier, index: true
      t.string :account_number
      t.timestamps null: false
    end

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps null: false
    end
  end
end
```

2.6 The `has_and_belongs_to_many` Association

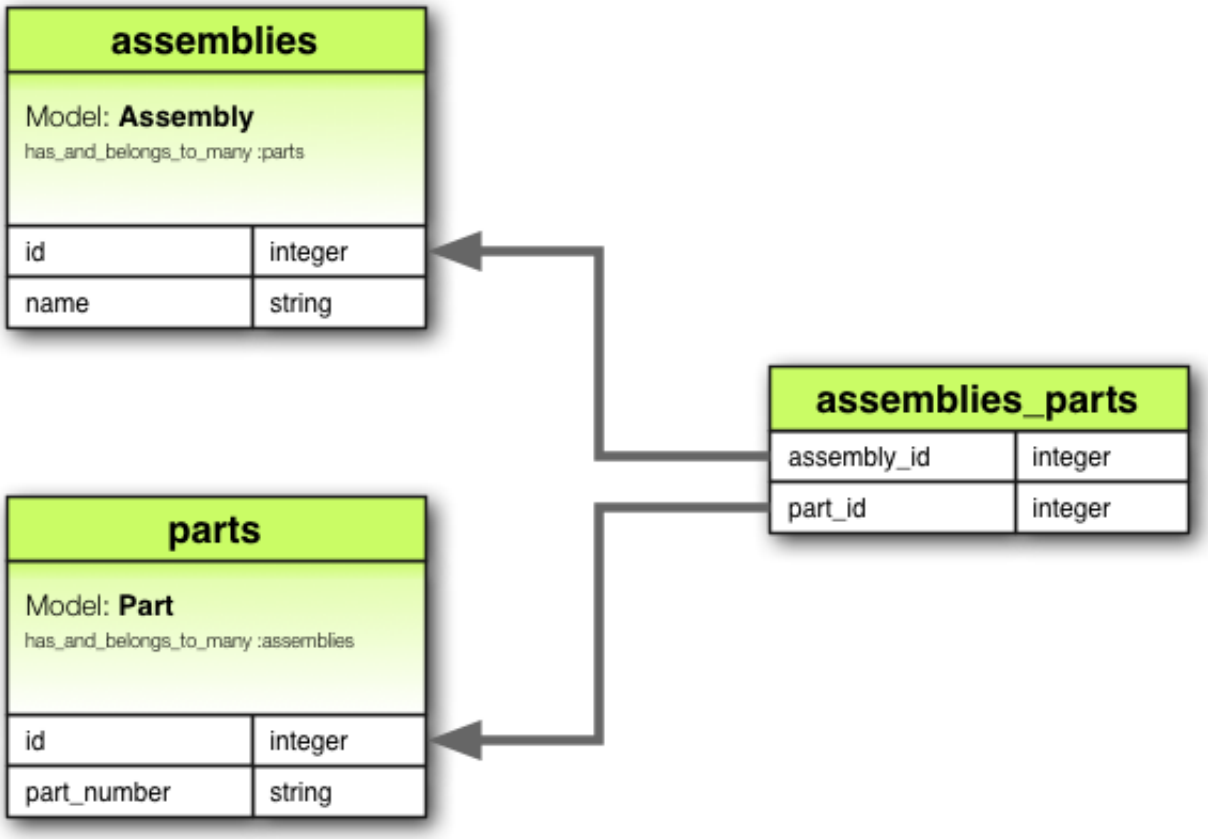
A `has_and_belongs_to_many` association creates a direct many-to-many connection with another model, with no intervening model. For example, if your application includes assemblies and parts, with each assembly having many parts and each part appearing in many assemblies, you could declare the models this way:




```
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
```

```
end
```




```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

The corresponding migration might look like this:

```
class CreateAssembliesAndParts < ActiveRecord::Migration
  def change
    create_table :assemblies do |t|
      t.string :name
      t.timestamps null: false
    end

    create_table :parts do |t|
```

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

    create_table :assemblies_parts, id: false do |t|
      t.belongs_to :assembly, index: true
      t.belongs_to :part, index: true
    end
  end
end
```

2.7 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. This suggests that the correct relationships are like this:

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration
  def change
    create_table :suppliers do |t|
      t.string  :name
      t.timestamps null: false
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps null: false
    end

    add_index :accounts, :supplier_id
  end
end
```

Using `t.integer :supplier_id` makes the foreign key naming obvious and explicit. In current versions of Rails, you can abstract away this implementation detail by using `t.references :supplier` instead.

2.8 Choosing Between `has_many :through` and `has_and_belongs_to_many`

Rails offers two different ways to declare a many-to-many relationship between models. The simpler way is to use `has_and_belongs_to_many`, which allows you to make the association directly:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

The second way to declare a many-to-many relationship is to use `has_many :through`. This makes the association indirectly, through a join model:

```
class Assembly < ActiveRecord::Base
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ActiveRecord::Base
  belongs_to :assembly
  belongs_to :part
end

class Part < ActiveRecord::Base
  has_many :manifests
  has_many :assemblies, through: :manifests
end
```

The simplest rule of thumb is that you should set up a `has_many :through` relationship if you need to work with the relationship model as an independent entity. If you don't need to do anything with the relationship model, it may be simpler to set up a `has_and_belongs_to_many` relationship (though you'll need to remember to create the joining table in the database).

You should use `has_many :through` if you need validations, callbacks, or extra attributes on the join model.

2.9 Polymorphic Associations

A slightly more advanced twist on associations is the *polymorphic association*. With polymorphic associations, a model can belong to more than one other model, on a single association. For example, you might have a picture model that belongs to either an employee model or a product model. Here's

how this could be declared:

```
class Picture < ActiveRecord::Base
  belongs_to :imageable, polymorphic: true
end

class Employee < ActiveRecord::Base
  has_many :pictures, as: :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, as: :imageable
end
```

You can think of a polymorphic `belongs_to` declaration as setting up an interface that any other model can use. From an instance of the `Employee` model, you can retrieve a collection of pictures: `@employee.pictures`.

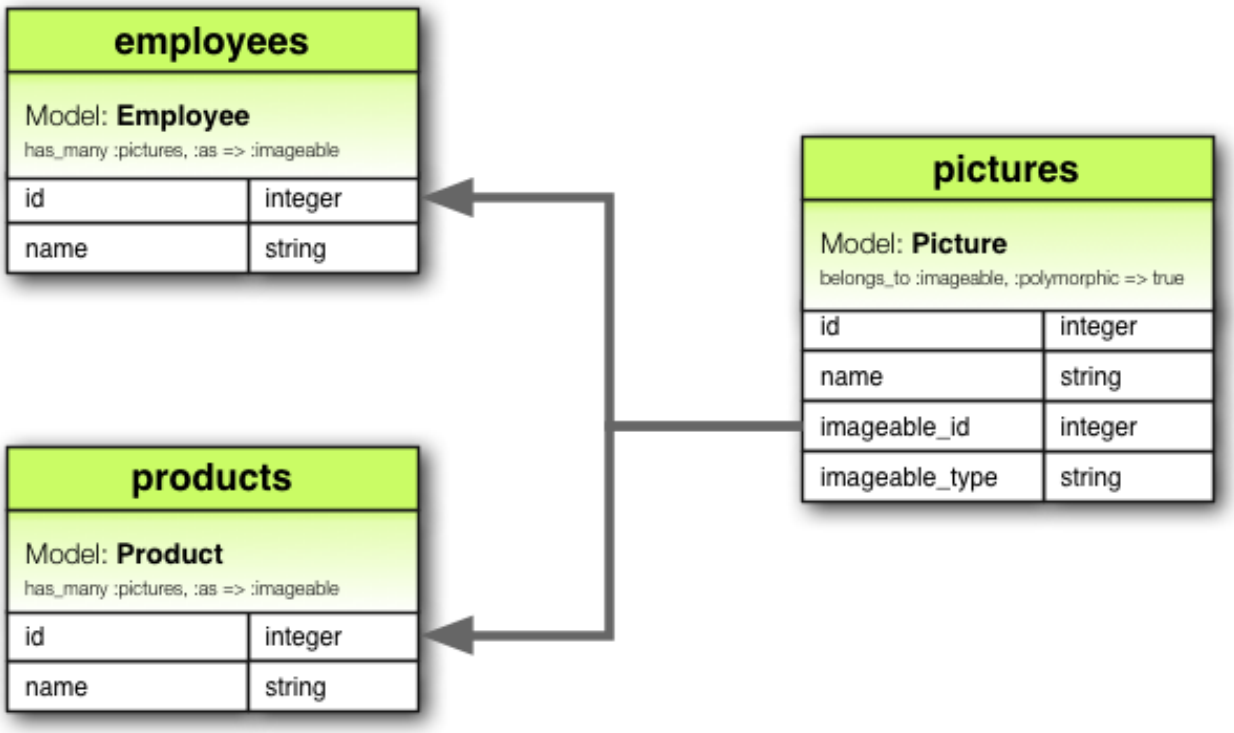
Similarly, you can retrieve `@product.pictures`.

If you have an instance of the `Picture` model, you can get to its parent via `@picture.imageable`. To make this work, you need to declare both a foreign key column and a type column in the model that declares the polymorphic interface:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps null: false
    end

    add_index :pictures, :imageable_id
  end
end
```

This migration can be simplified by using the `t.references` form:

```
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true, index: true
      t.timestamps null: false
    end
  end
end
```




```
class Picture < ActiveRecord::Base
  belongs_to :imageable, :polymorphic => true
end

class Employee < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end

class Product < ActiveRecord::Base
  has_many :pictures, :as => :imageable
end
```

2.10 Self Joins

In designing a data model, you will sometimes find a model that should have a relation to itself. For example, you may want to store all employees in a single database model, but be able to trace relationships such as between manager and subordinates. This situation can be modeled with self-joining associations:

```
class Employee < ActiveRecord::Base
  has_many :subordinates, class_name: "Employee",
                          foreign_key: "manager_id"

  belongs_to :manager, class_name: "Employee"
end
```

With this setup, you can retrieve `@employee.subordinates` and `@employee.manager`.

In your migrations/schema, you will add a references column to the model itself.

```
class CreateEmployees < ActiveRecord::Migration
  def change
    create_table :employees do |t|
      t.references :manager, index: true
      t.timestamps null: false
    end
  end
end
```

3 Tips, Tricks, and Warnings

Here are a few things you should know to make efficient use of Active Record associations in your Rails applications:

- Controlling caching
- Avoiding name collisions
- Updating the schema
- Controlling association scope
- Bi-directional associations

3.1 Controlling Caching

All of the association methods are built around caching, which keeps the result of the most recent query available for further operations. The cache is even shared across methods. For example:

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
customer.orders.empty?          # uses the cached copy of orders
```

But what if you want to reload the cache, because data might have been changed by some other part of the application? Just pass `true` to the association call:

```
customer.orders                 # retrieves orders from the database
customer.orders.size            # uses the cached copy of orders
```

```
customer.orders(true).empty?    # discards the cached copy of orders
                                # and goes back to the database
```

3.2 Avoiding Name Collisions

You are not free to use just any name for your associations. Because creating an association adds a method with that name to the model, it is a bad idea to give an association a name that is already used for an instance method of `ActiveRecord::Base`. The association method would override the base method and break things. For instance, `attributes` or `connection` are bad names for associations.

3.3 Updating the Schema

Associations are extremely useful, but they are not magic. You are responsible for maintaining your database schema to match your associations. In practice, this means two things, depending on what sort of associations you are creating. For `belongs_to` associations you need to create foreign keys, and for `has_and_belongs_to_many` associations you need to create the appropriate join table.

3.3.1 Creating Foreign Keys for `belongs_to` Associations

When you declare a `belongs_to` association, you need to create foreign keys as appropriate. For example, consider this model:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

This declaration needs to be backed up by the proper foreign key declaration on the orders table:

```
class CreateOrders < ActiveRecord::Migration
  def change
    create_table :orders do |t|
      t.datetime :order_date
      t.string   :order_number
      t.integer  :customer_id
    end

    add_index :orders, :customer_id
  end
end
```

If you create an association some time after you build the underlying model, you need to remember to create an `add_column` migration to provide the necessary foreign key.

3.3.2 Creating Join Tables for `has_and_belongs_to_many` Associations

If you create a `has_and_belongs_to_many` association, you need to explicitly create the joining table. Unless the name of the join table is explicitly specified by using the `:join_table` option, Active Record

creates the name by using the lexical order of the class names. So a join between customer and order models will give the default join table name of "customers_orders" because "c" outranks "o" in lexical ordering.

> The precedence between model names is calculated using the `<` operator for `String`. This means that if the strings are of different lengths, and the strings are equal when compared up to the shortest length, then the longer string is considered of higher lexical precedence than the shorter one. For example, one would expect the tables "paper_boxes" and "papers" to generate a join table name of "papers_paper_boxes" because of the length of the name "paper_boxes", but it in fact generates a join table name of "paper_boxes_papers" (because the underscore '_' *is lexicographically _less* than 's' in common encodings).

Whatever the name, you must manually generate the join table with an appropriate migration. For example, consider these associations:

```
class Assembly < ActiveRecord::Base
  has_and_belongs_to_many :parts
end

class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

These need to be backed up by a migration to create the `assemblies_parts` table. This table should be created without a primary key:

```
class CreateAssembliesPartsJoinTable < ActiveRecord::Migration
  def change
    create_table :assemblies_parts, id: false do |t|
      t.integer :assembly_id
      t.integer :part_id
    end

    add_index :assemblies_parts, :assembly_id
    add_index :assemblies_parts, :part_id
  end
end
```

We pass `id: false` to `create_table` because that table does not represent a model. That's required for the association to work properly. If you observe any strange behavior in a `has_and_belongs_to_many` association like mangled models IDs, or exceptions about conflicting IDs, chances are you forgot that bit.

3.4 Controlling Association Scope

By default, associations look for objects only within the current module's scope. This can be important

when you declare Active Record models within a module. For example:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end

    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

This will work fine, because both the `Supplier` and the `Account` class are defined within the same scope. But the following will *not* work, because `Supplier` and `Account` are defined in different scopes:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier
    end
  end
end
```

To associate a model with a model in a different namespace, you must specify the complete class name in your association declaration:

```
module MyApplication
  module Business
    class Supplier < ActiveRecord::Base
       has_one :account,
        class_name: "MyApplication::Billing::Account"
    end
  end

  module Billing
    class Account < ActiveRecord::Base
       belongs_to :supplier,
        class_name: "MyApplication::Business::Supplier"
    end
  end
end
```

3.5 Bi-directional Associations

It's normal for associations to work in two directions, requiring declaration on two different models:

```
class Customer < ActiveRecord::Base
  has_many :orders
end

class Order < ActiveRecord::Base
  belongs_to :customer
end
```

By default, Active Record doesn't know about the connection between these associations. This can lead to two copies of an object getting out of sync:

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => false
```

This happens because c and o.customer are two different in-memory representations of the same data, and neither one is automatically refreshed from changes to the other. Active Record provides the `:inverse_of` option so that you can inform it of these relations:

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

With these changes, Active Record will only load one copy of the customer object, preventing inconsistencies and making your application more efficient:

```
c = Customer.first
o = c.orders.first
c.first_name == o.customer.first_name # => true
c.first_name = 'Manny'
c.first_name == o.customer.first_name # => true
```

There are a few limitations to `inverse_of` support:

They do not work with `:through` associations.
They do not work with `:polymorphic` associations.
They do not work with `:as` associations.
For `belongs_to` associations, `has_many` inverse associations are ignored.

Every association will attempt to automatically find the inverse association and set the `:inverse_of` option heuristically (based on the association name). Most associations with standard names will be supported. However, associations that contain the following options will not have their inverses set automatically:

:conditions
:through
:polymorphic
:foreign_key

4 Detailed Association Reference

The following sections give the details of each type of association, including the methods that they add and the options that you can use when declaring an association.

4.1 `belongs_to` Association Reference

The `belongs_to` association creates a one-to-one match with another model. In database terms, this association says that this class contains the foreign key. If the other class contains the foreign key, then you should use `has_one` instead.

4.1.1 Methods Added by `belongs_to`

When you declare a `belongs_to` association, the declaring class automatically gains five methods related to the association:

```
association(force_reload = false)
association=(associate)
build_association(attributes = {})
create_association(attributes = {})
create_association!(attributes = {})
```

In all of these methods, `association` is replaced with the symbol passed as the first argument to `belongs_to`. For example, given the declaration:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
```

Each instance of the `Order` model will have these methods:

```
customer
customer=
build_customer
create_customer
create_customer!
```

When initializing a new `has_one` or `belongs_to` association you must use the `build_` prefix to build the association, rather than the `association.build` method that would be used for `has_many` or `has_and_belongs_to_many` associations. To create one, use the `create_` prefix.

4.1.1.1 `association(force_reload = false)`

The `association` method returns the associated object, if any. If no associated object is found, it returns `nil`.

```
@customer = @order.customer
```

If the associated object has already been retrieved from the database for this object, the cached version will be returned. To override this behavior (and force a database read), pass `true` as the `force_reload` argument.

4.1.1.2 `association=(associate)`

The `association=` method assigns an associated object to this object. Behind the scenes, this means extracting the primary key from the associate object and setting this object's foreign key to the same value.

```
@order.customer = @customer
```

4.1.1.3 `build_association(attributes = {})`

The `build_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through this object's foreign key will be set, but the associated object will *not* yet be saved.

```
@customer = @order.build_customer(customer_number: 123,
                                  customer_name: "John Doe")
```

4.1.1.4 `create_association(attributes = {})`

The `create_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through this object's foreign key will be set, and, once it passes all of the validations specified on the associated model, the associated object *will* be saved.

```
@customer = @order.create_customer(customer_number: 123,
                                   customer_name: "John Doe")
```

4.1.1.5 `create_association!(attributes = {})`

Does the same as `create_association` above, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

4.1.2 Options for `belongs_to`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `belongs_to` association reference. Such customizations can easily be accomplished by passing options and scope blocks when you create the association. For example, this association uses two such options:

```
class Order < ActiveRecord::Base
  belongs_to :customer, dependent: :destroy,
    counter_cache: true
end
```

The `belongs_to` association supports these options:

```
:autosave
:class_name
:counter_cache
:dependent
:foreign_key
:inverse_of
:polymorphic
:touch
:validate
```

4.1.2.1 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.1.2.2 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if an order belongs to a customer, but the actual name of the model containing customers is `Patron`, you'd set things up this way:

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron"
end
```

4.1.2.3 `:counter_cache`

The `:counter_cache` option can be used to make finding the number of belonging objects more efficient. Consider these models:

```
class Order < ActiveRecord::Base
  belongs_to :customer
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

With these declarations, asking for the value of `@customer.orders.size` requires making a call to the database to perform a `COUNT(*)` query. To avoid this call, you can add a counter cache to the *belonging* model:

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: true
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

With this declaration, Rails will keep the cache value up to date, and then return that value in response to the `size` method.

Although the `:counter_cache` option is specified on the model that includes the `belongs_to` declaration, the actual column must be added to the *associated* model. In the case above, you would need to add a column named `orders_count` to the `Customer` model. You can override the default column name if you need to:

```
class Order < ActiveRecord::Base
  belongs_to :customer, counter_cache: :count_of_orders
end
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Counter cache columns are added to the containing model's list of read-only attributes through `attr_readonly`.

4.1.2.4 `:dependent`

If you set the `:dependent` option to:

`:destroy`, when the object is destroyed, `destroy` will be called on its associated objects.

`:delete`, when the object is destroyed, all its associated objects will be deleted directly from the database without calling their `destroy` method.

You should not specify this option on a `belongs_to` association that is connected with a `has_many` association on the other class. Doing so can lead to orphaned records in your database.

4.1.2.5 `:foreign_key`

By convention, Rails assumes that the column used to hold the foreign key on this model is the name of the association with the suffix `_id` added. The `:foreign_key` option lets you set the name of the foreign key directly:

```
class Order < ActiveRecord::Base
  belongs_to :customer, class_name: "Patron",
                        foreign_key: "patron_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.1.2.6 `:inverse_of`

The `:inverse_of` option specifies the name of the `has_many` or `has_one` association that is the inverse of this association. Does not work in combination with the `:polymorphic` options.

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

4.1.2.7 `:polymorphic`

Passing `true` to the `:polymorphic` option indicates that this is a polymorphic association. Polymorphic associations were discussed in detail **earlier in this guide**.

4.1.2.8 `:touch`

If you set the `:touch` option to `:true`, then the `updated_at` or `updated_on` timestamp on the associated object will be set to the current time whenever this object is saved or destroyed:

```
class Order < ActiveRecord::Base
  belongs_to :customer, touch: true
```

```
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

In this case, saving or destroying an order will update the timestamp on the associated customer. You can also specify a particular timestamp attribute to update:

```
class Order < ActiveRecord::Base
  belongs_to :customer, touch: :orders_updated_at
end
```

4.1.2.9 `:validate`

If you set the `:validate` option to `true`, then associated objects will be validated whenever you save this object. By default, this is `false`: associated objects will not be validated when this object is saved.

4.1.3 Scopes for `belongs_to`

There may be times when you wish to customize the query used by `belongs_to`. Such customizations can be achieved via a scope block. For example:

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true },
                        dependent: :destroy
end
```

You can use any of the standard **querying methods** inside the scope block. The following ones are discussed below:

```
where
includes
readonly
select
```

4.1.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Order < ActiveRecord::Base
  belongs_to :customer, -> { where active: true }
end
```

4.1.3.2 `includes`

You can use the `includes` method to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class LineItem < ActiveRecord::Base
  belongs_to :order
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

If you frequently retrieve customers directly from line items (`@line_item.order.customer`), then you can make your code somewhat more efficient by including customers in the association from line items to orders:

```
class LineItem < ActiveRecord::Base
  belongs_to :order, -> { includes :customer }
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class Customer < ActiveRecord::Base
  has_many :orders
end
```

There's no need to use `includes` for immediate associations - that is, if you have `Order belongs_to :customer`, then the customer is eager-loaded automatically when it's needed.

4.1.3.3 `readonly`

If you use `readonly`, then the associated object will be read-only when retrieved via the association.

4.1.3.4 `select`

The `select` method lets you override the SQL `SELECT` clause that is used to retrieve data about the associated object. By default, Rails retrieves all columns.

If you use the `select` method on a `belongs_to` association, you should also set the `:foreign_key` option to guarantee the correct results.

4.1.4 Do Any Associated Objects Exist?

You can see if any associated objects exist by using the `association.nil?` method:

```
if @order.customer.nil?
  @msg = "No customer found for this order"
end
```

4.1.5 When are Objects Saved?

Assigning an object to a `belongs_to` association does *not* automatically save the object. It does not save the associated object either.

4.2 `has_one` Association Reference

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.

4.2.1 Methods Added by `has_one`

When you declare a `has_one` association, the declaring class automatically gains five methods related to the association:

```
association(force_reload = false)
association=(associate)
build_association(attributes = {})
create_association(attributes = {})
create_association!(attributes = {})
```

In all of these methods, `association` is replaced with the symbol passed as the first argument to `has_one`. For example, given the declaration:

```
class Supplier < ActiveRecord::Base
  has_one :account
end
```

Each instance of the `Supplier` model will have these methods:

```
account
account=
build_account
create_account
create_account!
```

When initializing a new `has_one` or `belongs_to` association you must use the `build_` prefix to build the association, rather than the `association.build` method that would be used for `has_many` or

`has_and_belongs_to_many` associations. To create one, use the `create_` prefix.

4.2.1.1 `association(force_reload = false)`

The `association` method returns the associated object, if any. If no associated object is found, it returns `nil`.

```
@account = @supplier.account
```

If the associated object has already been retrieved from the database for this object, the cached version will be returned. To override this behavior (and force a database read), pass `true` as the `force_reload` argument.

4.2.1.2 `association=(associate)`

The `association=` method assigns an associated object to this object. Behind the scenes, this means extracting the primary key from this object and setting the associate object's foreign key to the same value.

```
@supplier.account = @account
```

4.2.1.3 `build_association(attributes = {})`

The `build_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through its foreign key will be set, but the associated object will *not* yet be saved.

```
@account = @supplier.build_account(terms: "Net 30")
```

4.2.1.4 `create_association(attributes = {})`

The `create_association` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through its foreign key will be set, and, once it passes all of the validations specified on the associated model, the associated object *will* be saved.

```
@account = @supplier.create_account(terms: "Net 30")
```

4.2.1.5 `create_association!(attributes = {})`

Does the same as `create_association` above, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

4.2.2 Options for `has_one`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `has_one` association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this association uses two such options:

```
class Supplier < ActiveRecord::Base
  has_one :account, class_name: "Billing", dependent: :nullify
end
```

The `has_one` association supports these options:

```
:as
:autosave
:class_name
:dependent
:foreign_key
:inverse_of
:primary_key
:source
:source_type
:through
:validate
```

4.2.2.1 `:as`

Setting the `:as` option indicates that this is a polymorphic association. Polymorphic associations were discussed in detail **earlier in this guide**.

4.2.2.2 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.2.2.3 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if a supplier has an account, but the actual name of the model containing accounts is `Billing`, you'd set things up this way:

```
class Supplier < ActiveRecord::Base
  has_one :account, class_name: "Billing"
end
```

4.2.2.4 `:dependent`

Controls what happens to the associated object when its owner is destroyed:

`:destroy` causes the associated object to also be destroyed
`:delete` causes the associated object to be deleted directly from the database (so callbacks will not execute)
`:nullify` causes the foreign key to be set to `NULL`. Callbacks are not executed.
`:restrict_with_exception` causes an exception to be raised if there is an associated record
`:restrict_with_error` causes an error to be added to the owner if there is an associated object

It's necessary not to set or leave `:nullify` option for those associations that have `NOT NULL` database constraints. If you don't set `dependent` to destroy such associations you won't be able to change the associated object because initial associated object foreign key will be set to unallowed `NULL` value.

4.2.2.5 `:foreign_key`

By convention, Rails assumes that the column used to hold the foreign key on the other model is the name of this model with the suffix `_id` added. The `:foreign_key` option lets you set the name of the foreign key directly:

```
class Supplier < ActiveRecord::Base
  has_one :account, foreign_key: "supp_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.2.2.6 `:inverse_of`

The `:inverse_of` option specifies the name of the `belongs_to` association that is the inverse of this association. Does not work in combination with the `:through` or `:as` options.

```
class Supplier < ActiveRecord::Base
  has_one :account, inverse_of: :supplier
end

class Account < ActiveRecord::Base
  belongs_to :supplier, inverse_of: :account
end
```

4.2.2.7 `:primary_key`

By convention, Rails assumes that the column used to hold the primary key of this model is `id`. You can override this and explicitly specify the primary key with the `:primary_key` option.

4.2.2.8 `:source`

The `:source` option specifies the source association name for a `has_one :through` association.

4.2.2.9 `:source_type`

The `:source_type` option specifies the source association type for a `has_one :through` association that proceeds through a polymorphic association.

4.2.2.10 `:through`

The `:through` option specifies a join model through which to perform the query. `has_one :through` associations were discussed in detail **earlier in this guide**.

4.2.2.11 `:validate`

If you set the `:validate` option to `true`, then associated objects will be validated whenever you save this object. By default, this is `false`: associated objects will not be validated when this object is saved.

4.2.3 Scopes for `has_one`

There may be times when you wish to customize the query used by `has_one`. Such customizations can be achieved via a scope block. For example:

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where active: true }
end
```

You can use any of the standard **querying methods** inside the scope block. The following ones are discussed below:

```
where
includes
readonly
select
```

4.2.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { where "confirmed = 1" }
end
```

4.2.3.2 `includes`

You can use the `includes` method to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class Supplier < ActiveRecord::Base
  has_one :account
end

class Account < ActiveRecord::Base
```

```
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

If you frequently retrieve representatives directly from suppliers (`@supplier.account.representative`), then you can make your code somewhat more efficient by including representatives in the association from suppliers to accounts:

```
class Supplier < ActiveRecord::Base
  has_one :account, -> { includes :representative }
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  belongs_to :representative
end

class Representative < ActiveRecord::Base
  has_many :accounts
end
```

4.2.3.3 `readonly`

If you use the `readonly` method, then the associated object will be read-only when retrieved via the association.

4.2.3.4 `select`

The `select` method lets you override the SQL `SELECT` clause that is used to retrieve data about the associated object. By default, Rails retrieves all columns.

4.2.4 Do Any Associated Objects Exist?

You can see if any associated objects exist by using the `association.nil?` method:

```
if @supplier.account.nil?
  @msg = "No account found for this supplier"
end
```

4.2.5 When are Objects Saved?

When you assign an object to a `has_one` association, that object is automatically saved (in order to update its foreign key). In addition, any object being replaced is also automatically saved, because its foreign key will change too.

If either of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the `has_one` association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved. They will automatically when the parent object is saved.

If you want to assign an object to a `has_one` association without saving the object, use the `association.build` method.

4.3 `has_many` Association Reference

The `has_many` association creates a one-to-many relationship with another model. In database terms, this association says that the other class will have a foreign key that refers to instances of this class.

4.3.1 Methods Added by `has_many`

When you declare a `has_many` association, the declaring class automatically gains 16 methods related to the association:

```
collection(force_reload = false)
collection<<(object, ...)
collection.delete(object, ...)
collection.destroy(object, ...)
collection=(objects)
collection_singular_ids
collection_singular_ids=(ids)
collection.clear
collection.empty?
collection.size
collection.find(...)
collection.where(...)
collection.exists?(...)
collection.build(attributes = {}, ...)
collection.create(attributes = {})
collection.create!(attributes = {})
```

In all of these methods, `collection` is replaced with the symbol passed as the first argument to `has_many`, and `collection_singular` is replaced with the singularized version of that symbol. For example, given the declaration:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

Each instance of the `Customer` model will have these methods:

```
orders(force_reload = false)
orders<<(object, ...)
orders.delete(object, ...)
orders.destroy(object, ...)
orders=(objects)
order_ids
order_ids=(ids)
orders.clear
orders.empty?
orders.size
orders.find(...)
orders.where(...)
orders.exists?(...)
orders.build(attributes = {}, ...)
orders.create(attributes = {})
orders.create!(attributes = {})
```

4.3.1.1 `collection(force_reload = false)`

The `collection` method returns an array of all of the associated objects. If there are no associated objects, it returns an empty array.

```
@orders = @customer.orders
```

4.3.1.2 `collection<<(object, ...)`

The `collection<<` method adds one or more objects to the collection by setting their foreign keys to the primary key of the calling model.

```
@customer.orders << @order1
```

4.3.1.3 `collection.delete(object, ...)`

The `collection.delete` method removes one or more objects from the collection by setting their foreign keys to `NULL`.

```
@customer.orders.delete(@order1)
```

Additionally, objects will be destroyed if they're associated with `dependent: :destroy`, and deleted if they're associated with `dependent: :delete_all`.

4.3.1.4 `collection.destroy(object, ...)`

The `collection.destroy` method removes one or more objects from the collection by running `destroy` on each object.

```
@customer.orders.destroy(@order1)
```

Objects will *always* be removed from the database, ignoring the `:dependent` option.

4.3.1.5 `collection=(objects)`

The `collection=` method makes the collection contain only the supplied objects, by adding and deleting as appropriate.

4.3.1.6 `collection_singular_ids`

The `collection_singular_ids` method returns an array of the ids of the objects in the collection.

```
@order_ids = @customer.order_ids
```

4.3.1.7 `collection_singular_ids=(ids)`

The `collection_singular_ids=` method makes the collection contain only the objects identified by the supplied primary key values, by adding and deleting as appropriate.

4.3.1.8 `collection.clear`

The `collection.clear` method removes all objects from the collection according to the strategy specified by the `dependent` option. If no option is given, it follows the default strategy. The default strategy for `has_many :through` associations is `delete_all`, and for `has_many` associations is to set the foreign keys to `NULL`.

```
@customer.orders.clear
```

Objects will be delete if they're associated with `dependent: :destroy`, just like `dependent: :delete_all`.

4.3.1.9 `collection.empty?`

The `collection.empty?` method returns `true` if the collection does not contain any associated objects.

```
<% if @customer.orders.empty? %>
  No Orders Found
<% end %>
```

4.3.1.10 `collection.size`

The `collection.size` method returns the number of objects in the collection.

```
@order_count = @customer.orders.size
```

4.3.1.11 `collection.find(...)`

The `collection.find` method finds objects within the collection. It uses the same syntax and options as `ActiveRecord::Base.find`.

```
@open_orders = @customer.orders.find(1)
```

4.3.1.12 `collection.where(...)`

The `collection.where` method finds objects within the collection based on the conditions supplied but the objects are loaded lazily meaning that the database is queried only when the object(s) are accessed.

```
@open_orders = @customer.orders.where(open: true) # No query yet
@open_order = @open_orders.first # Now the database will be queried
```

4.3.1.13 `collection.exists?(...)`

The `collection.exists?` method checks whether an object meeting the supplied conditions exists in the collection. It uses the same syntax and options as `ActiveRecord::Base.exists?`.

4.3.1.14 `collection.build(attributes = {}, ...)`

The `collection.build` method returns one or more new objects of the associated type. These objects will be instantiated from the passed attributes, and the link through their foreign key will be created, but the associated objects will *not* yet be saved.

```
@order = @customer.orders.build(order_date: Time.now,
                                order_number: "A12345")
```

4.3.1.15 `collection.create(attributes = {})`

The `collection.create` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through its foreign key will be created, and, once it passes all of the validations specified on the associated model, the associated object *will* be saved.

```
@order = @customer.orders.create(order_date: Time.now,
                                 order_number: "A12345")
```

4.3.1.16 `collection.create!(attributes = {})`

Does the same as `collection.create` above, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

4.3.2 Options for `has_many`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `has_many` association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this association uses two such options:

```
class Customer < ActiveRecord::Base
  has_many :orders, dependent: :delete_all, validate: :false
end
```

The `has_many` association supports these options:

```
:as
:autosave
:class_name
:dependent
:foreign_key
:inverse_of
:primary_key
:source
:source_type
:through
:validate
```

4.3.2.1 `:as`

Setting the `:as` option indicates that this is a polymorphic association, as discussed **earlier in this guide**.

4.3.2.2 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.3.2.3 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if a customer has many orders, but the actual name of the model containing orders is `Transaction`, you'd set things up this way:

```
class Customer < ActiveRecord::Base
  has_many :orders, class_name: "Transaction"
end
```

4.3.2.4 `:dependent`

Controls what happens to the associated objects when their owner is destroyed:

`:destroy` causes all the associated objects to also be destroyed
`:delete_all` causes all the associated objects to be deleted directly from the database (so callbacks will not execute)
`:nullify` causes the foreign keys to be set to `NULL`. Callbacks are not executed.
`:restrict_with_exception` causes an exception to be raised if there are any associated records
`:restrict_with_error` causes an error to be added to the owner if there are any associated objects

4.3.2.5 `:foreign_key`

By convention, Rails assumes that the column used to hold the foreign key on the other model is the name of this model with the suffix `_id` added. The `:foreign_key` option lets you set the name of the foreign key directly:

```
class Customer < ActiveRecord::Base
  has_many :orders, foreign_key: "cust_id"
end
```

In any case, Rails will not create foreign key columns for you. You need to explicitly define them as part of your migrations.

4.3.2.6 `:inverse_of`

The `:inverse_of` option specifies the name of the `belongs_to` association that is the inverse of this association. Does not work in combination with the `:through` or `:as` options.

```
class Customer < ActiveRecord::Base
  has_many :orders, inverse_of: :customer
end

class Order < ActiveRecord::Base
  belongs_to :customer, inverse_of: :orders
end
```

4.3.2.7 `:primary_key`

By convention, Rails assumes that the column used to hold the primary key of the association is `id`. You can override this and explicitly specify the primary key with the `:primary_key` option.

Let's say that `users` table has `id` as the primary_key but it also has `guid` column. And the requirement is that `todos` table should hold `guid` column value and not `id` value. This can be achieved like this

```
class User < ActiveRecord::Base
  has_many :todos, primary_key: :guid
end
```

Now if we execute `@user.todos.create` then `@todo` record will have `user_id` value as the `guid` value of `@user`.

4.3.2.8 `:source`

The `:source` option specifies the source association name for a `has_many :through` association. You only need to use this option if the name of the source association cannot be automatically inferred from the association name.

4.3.2.9 `:source_type`

The `:source_type` option specifies the source association type for a `has_many :through` association that proceeds through a polymorphic association.

4.3.2.10 `:through`

The `:through` option specifies a join model through which to perform the query. `has_many :through` associations provide a way to implement many-to-many relationships, as discussed **earlier in this guide**.

4.3.2.11 `:validate`

If you set the `:validate` option to `false`, then associated objects will not be validated whenever you save this object. By default, this is `true`: associated objects will be validated when this object is saved.

4.3.3 Scopes for `has_many`

There may be times when you wish to customize the query used by `has_many`. Such customizations can be achieved via a scope block. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { where processed: true }
end
```

You can use any of the standard **querying methods** inside the scope block. The following ones are discussed below:

```
where
extending
group
includes
limit
offset
order
readonly
select
```

```
uniq
```

4.3.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, -> { where "confirmed = 1" },
    class_name: "Order"
end
```

You can also set conditions via a hash:

```
class Customer < ActiveRecord::Base
  has_many :confirmed_orders, -> { where confirmed: true },
                              class_name: "Order"
end
```

If you use a hash-style `where` option, then record creation via this association will be automatically scoped using the hash. In this case, using `@customer.confirmed_orders.create` or `@customer.confirmed_orders.build` will create orders where the confirmed column has the value `true`.

4.3.3.2 `extending`

The `extending` method specifies a named module to extend the association proxy. Association extensions are discussed in detail **later in this guide**.

4.3.3.3 `group`

The `group` method supplies an attribute name to group the result set by, using a `GROUP BY` clause in the finder SQL.

```
class Customer < ActiveRecord::Base
  has_many :line_items, -> { group 'orders.id' },
                        through: :orders
end
```

4.3.3.4 `includes`

You can use the `includes` method to specify second-order associations that should be eager-loaded when this association is used. For example, consider these models:

```
class Customer < ActiveRecord::Base
  has_many :orders
end
```

```
class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

If you frequently retrieve line items directly from customers (`@customer.orders.line_items`), then you can make your code somewhat more efficient by including line items in the association from customers to orders:

```
class Customer < ActiveRecord::Base
  has_many :orders, -> { includes :line_items }
end

class Order < ActiveRecord::Base
  belongs_to :customer
  has_many :line_items
end

class LineItem < ActiveRecord::Base
  belongs_to :order
end
```

4.3.3.5 `limit`

The `limit` method lets you restrict the total number of objects that will be fetched through an association.

```
class Customer < ActiveRecord::Base
  has_many :recent_orders,
    -> { order('order_date desc').limit(100) },
    class_name: "Order",
end
```

4.3.3.6 `offset`

The `offset` method lets you specify the starting offset for fetching objects via an association. For example, `-> { offset(11) }` will skip the first 11 records.

4.3.3.7 `order`

The `order` method dictates the order in which associated objects will be received (in the syntax used by an SQL `ORDER BY` clause).

```
class Customer < ActiveRecord::Base
```

```
  has_many :orders, -> { order "date_confirmed DESC" }
end
```

4.3.3.8 `readonly`

If you use the `readonly` method, then the associated objects will be read-only when retrieved via the association.

4.3.3.9 `select`

The `select` method lets you override the SQL `SELECT` clause that is used to retrieve data about the associated objects. By default, Rails retrieves all columns.

> If you specify your own `select`, be sure to include the primary key and foreign key columns of the associated model. If you do not, Rails will throw an error.

4.3.3.10 `distinct`

Use the `distinct` method to keep the collection free of duplicates. This is mostly useful together with the `:through` option.

```
class Person < ActiveRecord::Base
  has_many :readings
  has_many :articles, through: :readings
end

person = Person.create(name: 'John')
article   = Article.create(name: 'a1')
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 5, name: "a1">, #<Article
id: 5, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 12, person_id: 5,
article_id: 5>, #<Reading id: 13, person_id: 5, article_id: 5>]
```

In the above case there are two readings and `person.articles` brings out both of them even though these records are pointing to the same article.

Now let's set `distinct`:

```
class Person
  has_many :readings
  has_many :articles, -> { distinct }, through: :readings
end

person = Person.create(name: 'Honda')
article   = Article.create(name: 'a1')
```

```
person.articles << article
person.articles << article
person.articles.inspect # => [#<Article id: 7, name: "a1">]
Reading.all.inspect  # => [#<Reading id: 16, person_id: 7,
article_id: 7>, #<Reading id: 17, person_id: 7, article_id: 7>]
```

In the above case there are still two readings. However `person.articles` shows only one article because the collection loads only unique records.

If you want to make sure that, upon insertion, all of the records in the persisted association are distinct (so that you can be sure that when you inspect the association that you will never find duplicate records), you should add a unique index on the table itself. For example, if you have a table named `person_articles` and you want to make sure all the articles are unique, you could add the following in a migration:

```
add_index :person_articles, :article, unique: true
```

Note that checking for uniqueness using something like `include?` is subject to race conditions. Do not attempt to use `include?` to enforce distinctness in an association. For instance, using the article example from above, the following code would be racy because multiple users could be attempting this at the same time:

```
person.articles << article unless person.articles.include?(article)
```

4.3.4 When are Objects Saved?

When you assign an object to a `has_many` association, that object is automatically saved (in order to update its foreign key). If you assign multiple objects in one statement, then they are all saved.

If any of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the `has_many` association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved when they are added. All unsaved members of the association will automatically be saved when the parent is saved.

If you want to assign an object to a `has_many` association without saving the object, use the `collection.build` method.

4.4 `has_and_belongs_to_many` Association Reference

The `has_and_belongs_to_many` association creates a many-to-many relationship with another model. In database terms, this associates two classes via an intermediate join table that includes foreign keys referring to each of the classes.

4.4.1 Methods Added by `has_and_belongs_to_many`

When you declare a `has_and_belongs_to_many` association, the declaring class automatically gains 16 methods related to the association:

```
collection(force_reload = false)
collection<<(object, ...)
collection.delete(object, ...)
collection.destroy(object, ...)
collection=(objects)
collection_singular_ids
collection_singular_ids=(ids)
collection.clear
collection.empty?
collection.size
collection.find(...)
collection.where(...)
collection.exists?(...)
collection.build(attributes = {})
collection.create(attributes = {})
collection.create!(attributes = {})
```

In all of these methods, `collection` is replaced with the symbol passed as the first argument to `has_and_belongs_to_many`, and `collection_singular` is replaced with the singularized version of that symbol. For example, given the declaration:

```
class Part < ActiveRecord::Base
  has_and_belongs_to_many :assemblies
end
```

Each instance of the `Part` model will have these methods:

```
assemblies(force_reload = false)
assemblies<<(object, ...)
assemblies.delete(object, ...)
assemblies.destroy(object, ...)
assemblies=(objects)
assembly_ids
assembly_ids=(ids)
assemblies.clear
assemblies.empty?
assemblies.size
assemblies.find(...)
assemblies.where(...)
assemblies.exists?(...)
assemblies.build(attributes = {}, ...)
assemblies.create(attributes = {})
assemblies.create!(attributes = {})
```

4.4.1.1 Additional Column Methods

If the join table for a `has_and_belongs_to_many` association has additional columns beyond the two foreign keys, these columns will be added as attributes to records retrieved via that association. Records returned with additional attributes will always be read-only, because Rails cannot save changes to those attributes.

> The use of extra attributes on the join table in a `has_and_belongs_to_many` association is deprecated. If you require this sort of complex behavior on the table that joins two models in a many-to-many relationship, you should use a `has_many :through` association instead of `has_and_belongs_to_many`.

4.4.1.2 `collection(force_reload = false)`

The `collection` method returns an array of all of the associated objects. If there are no associated objects, it returns an empty array.

```
@assemblies = @part.assemblies
```

4.4.1.3 `collection<<(object, ...)`

The `collection<<` method adds one or more objects to the collection by creating records in the join table.

```
@part.assemblies << @assembly1
```

This method is aliased as `collection.concat` and `collection.push`.

4.4.1.4 `collection.delete(object, ...)`

The `collection.delete` method removes one or more objects from the collection by deleting records in the join table. This does not destroy the objects.

```
@part.assemblies.delete(@assembly1)
```

> This does not trigger callbacks on the join records.

4.4.1.5 `collection.destroy(object, ...)`

The `collection.destroy` method removes one or more objects from the collection by running `destroy` on each record in the join table, including running callbacks. This does not destroy the objects.

```
@part.assemblies.destroy(@assembly1)
```

4.4.1.6 `collection=(objects)`

The `collection=` method makes the collection contain only the supplied objects, by adding and deleting as appropriate.

4.4.1.7 `collection_singular_ids`

The `collection_singular_ids` method returns an array of the ids of the objects in the collection.

```
@assembly_ids = @part.assembly_ids
```

4.4.1.8 `collection_singular_ids=(ids)`

The `collection_singular_ids=` method makes the collection contain only the objects identified by the supplied primary key values, by adding and deleting as appropriate.

4.4.1.9 `collection.clear`

The `collection.clear` method removes every object from the collection by deleting the rows from the joining table. This does not destroy the associated objects.

4.4.1.10 `collection.empty?`

The `collection.empty?` method returns `true` if the collection does not contain any associated objects.

```
<% if @part.assemblies.empty? %>
  This part is not used in any assemblies
<% end %>
```

4.4.1.11 `collection.size`

The `collection.size` method returns the number of objects in the collection.

```
@assembly_count = @part.assemblies.size
```

4.4.1.12 `collection.find(...)`

The `collection.find` method finds objects within the collection. It uses the same syntax and options as `ActiveRecord::Base.find`. It also adds the additional condition that the object must be in the collection.

```
@assembly = @part.assemblies.find(1)
```

4.4.1.13 `collection.where(...)`

The `collection.where` method finds objects within the collection based on the conditions supplied but the objects are loaded lazily meaning that the database is queried only when the object(s) are accessed. It also adds the additional condition that the object must be in the collection.

```
@new_assemblies = @part.assemblies.where("created_at > ?",
2.days.ago)
```

4.4.1.14 `collection.exists?(...)`

The `collection.exists?` method checks whether an object meeting the supplied conditions exists in the collection. It uses the same syntax and options as `ActiveRecord::Base.exists?`.

4.4.1.15 `collection.build(attributes = {})`

The `collection.build` method returns a new object of the associated type. This object will be instantiated from the passed attributes, and the link through the join table will be created, but the associated object will *not* yet be saved.

```
@assembly = @part.assemblies.build({assembly_name: "Transmission
housing"})
```

4.4.1.16 `collection.create(attributes = {})`

The `collection.create` method returns a new object of the associated type. This object will be instantiated from the passed attributes, the link through the join table will be created, and, once it passes all of the validations specified on the associated model, the associated object *will* be saved.

```
@assembly = @part.assemblies.create({assembly_name: "Transmission
housing"})
```

4.4.1.17 `collection.create!(attributes = {})`

Does the same as `collection.create`, but raises `ActiveRecord::RecordInvalid` if the record is invalid.

4.4.2 Options for `has_and_belongs_to_many`

While Rails uses intelligent defaults that will work well in most situations, there may be times when you want to customize the behavior of the `has_and_belongs_to_many` association reference. Such customizations can easily be accomplished by passing options when you create the association. For example, this association uses two such options:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { readonly },
```

```
                                        autosave: true
  end
```

The `has_and_belongs_to_many` association supports these options:

```
:association_foreign_key
:autosave
:class_name
:foreign_key
:join_table
:validate
```

4.4.2.1 `:association_foreign_key`

By convention, Rails assumes that the column in the join table used to hold the foreign key pointing to the other model is the name of that model with the suffix `_id` added. The `:association_foreign_key` option lets you set the name of the foreign key directly:

The `:foreign_key` and `:association_foreign_key` options are useful when setting up a many-to-many self-join. For example:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends,
      class_name: "User",
      foreign_key: "this_user_id",
      association_foreign_key: "other_user_id"
end
```

4.4.2.2 `:autosave`

If you set the `:autosave` option to `true`, Rails will save any loaded members and destroy members that are marked for destruction whenever you save the parent object.

4.4.2.3 `:class_name`

If the name of the other model cannot be derived from the association name, you can use the `:class_name` option to supply the model name. For example, if a part has many assemblies, but the actual name of the model containing assemblies is `Gadget`, you'd set things up this way:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, class_name: "Gadget"
end
```

4.4.2.4 `:foreign_key`

By convention, Rails assumes that the column in the join table used to hold the foreign key pointing to this model is the name of this model with the suffix `_id` added. The `:foreign_key` option lets you set the

name of the foreign key directly:

```
class User < ActiveRecord::Base
  has_and_belongs_to_many :friends,
      class_name: "User",
      foreign_key: "this_user_id",
      association_foreign_key: "other_user_id"
end
```

4.4.2.5 `:join_table`

If the default name of the join table, based on lexical ordering, is not what you want, you can use the `:join_table` option to override the default.

4.4.2.6 `:validate`

If you set the `:validate` option to `false`, then associated objects will not be validated whenever you save this object. By default, this is `true`: associated objects will be validated when this object is saved.

4.4.3 Scopes for `has_and_belongs_to_many`

There may be times when you wish to customize the query used by `has_and_belongs_to_many`. Such customizations can be achieved via a scope block. For example:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { where active: true }
end
```

You can use any of the standard **querying methods** inside the scope block. The following ones are discussed below:

```
where
extending
group
includes
limit
offset
order
readonly
select
uniq
```

4.4.3.1 `where`

The `where` method lets you specify the conditions that the associated object must meet.

```
class Parts < ActiveRecord::Base
```

```
  has_and_belongs_to_many :assemblies,
    -> { where "factory = 'Seattle'" }
end
```

You can also set conditions via a hash:

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { where factory: 'Seattle' }
end
```

If you use a hash-style `where`, then record creation via this association will be automatically scoped using the hash. In this case, using `@parts.assemblies.create` or `@parts.assemblies.build` will create orders where the `factory` column has the value "Seattle".

4.4.3.2 `extending`

The `extending` method specifies a named module to extend the association proxy. Association extensions are discussed in detail **later in this guide**.

4.4.3.3 `group`

The `group` method supplies an attribute name to group the result set by, using a `GROUP BY` clause in the finder SQL.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies, -> { group "factory" }
end
```

4.4.3.4 `includes`

You can use the `includes` method to specify second-order associations that should be eager-loaded when this association is used.

4.4.3.5 `limit`

The `limit` method lets you restrict the total number of objects that will be fetched through an association.

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { order("created_at DESC").limit(50) }
end
```

4.4.3.6 `offset`

The `offset` method lets you specify the starting offset for fetching objects via an association. For example, if you set `offset(11)`, it will skip the first 11 records.

4.4.3.7 `order`

The `order` method dictates the order in which associated objects will be received (in the syntax used by an SQL `ORDER BY` clause).

```
class Parts < ActiveRecord::Base
  has_and_belongs_to_many :assemblies,
    -> { order "assembly_name ASC" }
end
```

4.4.3.8 `readonly`

If you use the `readonly` method, then the associated objects will be read-only when retrieved via the association.

4.4.3.9 `select`

The `select` method lets you override the SQL `SELECT` clause that is used to retrieve data about the associated objects. By default, Rails retrieves all columns.

4.4.3.10 `uniq`

Use the `uniq` method to remove duplicates from the collection.

4.4.4 When are Objects Saved?

When you assign an object to a `has_and_belongs_to_many` association, that object is automatically saved (in order to update the join table). If you assign multiple objects in one statement, then they are all saved.

If any of these saves fails due to validation errors, then the assignment statement returns `false` and the assignment itself is cancelled.

If the parent object (the one declaring the `has_and_belongs_to_many` association) is unsaved (that is, `new_record?` returns `true`) then the child objects are not saved when they are added. All unsaved members of the association will automatically be saved when the parent is saved.

If you want to assign an object to a `has_and_belongs_to_many` association without saving the object, use the `collection.build` method.

4.5 Association Callbacks

Normal callbacks hook into the life cycle of Active Record objects, allowing you to work with those objects at various points. For example, you can use a `:before_save` callback to cause something to happen just before an object is saved.

Association callbacks are similar to normal callbacks, but they are triggered by events in the life cycle of a collection. There are four available association callbacks:

```
before_add
after_add
before_remove
after_remove
```

You define association callbacks by adding options to the association declaration. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders, before_add: :check_credit_limit

  def check_credit_limit(order)
    ...
  end
end
```

Rails passes the object being added or removed to the callback.

You can stack callbacks on a single event by passing them as an array:

```
class Customer < ActiveRecord::Base
  has_many :orders,
    before_add: [:check_credit_limit, :calculate_shipping_charges]

  def check_credit_limit(order)
    ...
  end

  def calculate_shipping_charges(order)
    ...
  end
end
```

If a `before_add` callback throws an exception, the object does not get added to the collection. Similarly, if a `before_remove` callback throws an exception, the object does not get removed from the collection.

4.6 Association Extensions

You're not limited to the functionality that Rails automatically builds into association proxy objects. You can also extend these objects through anonymous modules, adding new finders, creators, or other methods. For example:

```
class Customer < ActiveRecord::Base
  has_many :orders do
```

```
    def find_by_order_prefix(order_number)
      find_by(region_id: order_number[0..2])
    end
  end
end
```

If you have an extension that should be shared by many associations, you can use a named extension module. For example:

```
module FindRecentExtension
  def find_recent
    where("created_at > ?", 5.days.ago)
  end
end

class Customer < ActiveRecord::Base
  has_many :orders, -> { extending FindRecentExtension }
end

class Supplier < ActiveRecord::Base
  has_many :deliveries, -> { extending FindRecentExtension }
end
```

Extensions can refer to the internals of the association proxy using these three attributes of the `proxy_association` accessor:

- `proxy_association.owner` returns the object that the association is a part of.
- `proxy_association.reflection` returns the reflection object that describes the association.
- `proxy_association.target` returns the associated object for `belongs_to` or `has_one`, or the collection of associated objects for `has_many` or `has_and_belongs_to_many`.

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our **documentation contributions** section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check **Edge Guides** first to verify if the issues are already fixed or not on the master branch. Check the **Ruby on Rails Guides Guidelines** for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the **rubyonrails-docs mailing list**.