Advance Rails

1. **ActiveRecord associations**
    1. One-to-one, one-to-many
    2. Many-to-many
    3. ActiveRecord models and Ruby classes
    4. Call backs , Validations, Scope
2. **Fine-tuning controllers and view**
    1. Controller filters
    2. Redirecting requests
    3. Default and custom view helper methods
3. **Writing and processing HTML forms in Rails**
    1. Using form helper methods
    2. "Magic" field initialisation from instance variables
4. Image uploading
5. **Ruby/ Rails JSON**
    1. Ruby support for JSON
    2. HTTParty helps with communicating with RESTful services
    3. HTTParty gets Classy

1 Why Associations?

An *association* is a connection between two Active Record models.

Why do we need associations between models?

Since they make common operations simpler and easier in your code.

For example, consider a simple Rails application that includes a model for authors and a model for books.

Each author can have many books.

Without associations, the model declarations would look like this:

```
class Author < ApplicationRecord
end

class Book < ApplicationRecord
end
```

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

```
@book = Book.create(published_at: Time.now, author_id: 
@author.id)
```

Or consider deleting an author, and ensuring that all of its books get deleted as well:

```
@books = Book.where(author_id: @author.id)
@books.each do |book|
  book.destroy
end
@author.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 authors and books:

```
class Author < ApplicationRecord
  has_many :books, dependent: :destroy
end

class Book < ApplicationRecord
  belongs_to :author
end
```

With this change, creating a new book for a particular author is easier:

```
@book = @author.books.create(published_at: Time.now)
```

Deleting an author and all of its books is *much* easier:

```
@author.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

Rails supports six types of associations:

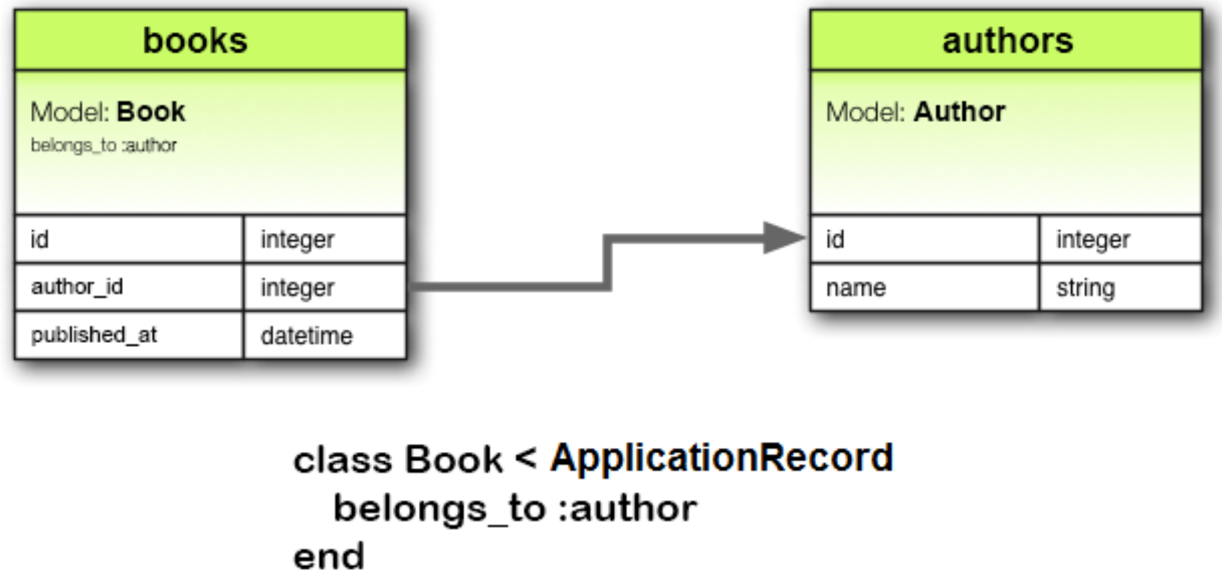
- `belongs_to`
- `has_one`
- `has_many`
- `has_many :through`
- `has_one :through`
- `has_and_belongs_to_many`

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.

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 authors and books, and each book can be assigned to exactly one author, you'd declare the book model this way:

```
class Book < ApplicationRecord
  belongs_to :author
end
```



Note:

`belongs_to` associations *must* use the singular term. If you used the pluralized form in the above example for the `author` association in the `Book` model, you would be told that there was an "uninitialized constant Book::Authors". 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.

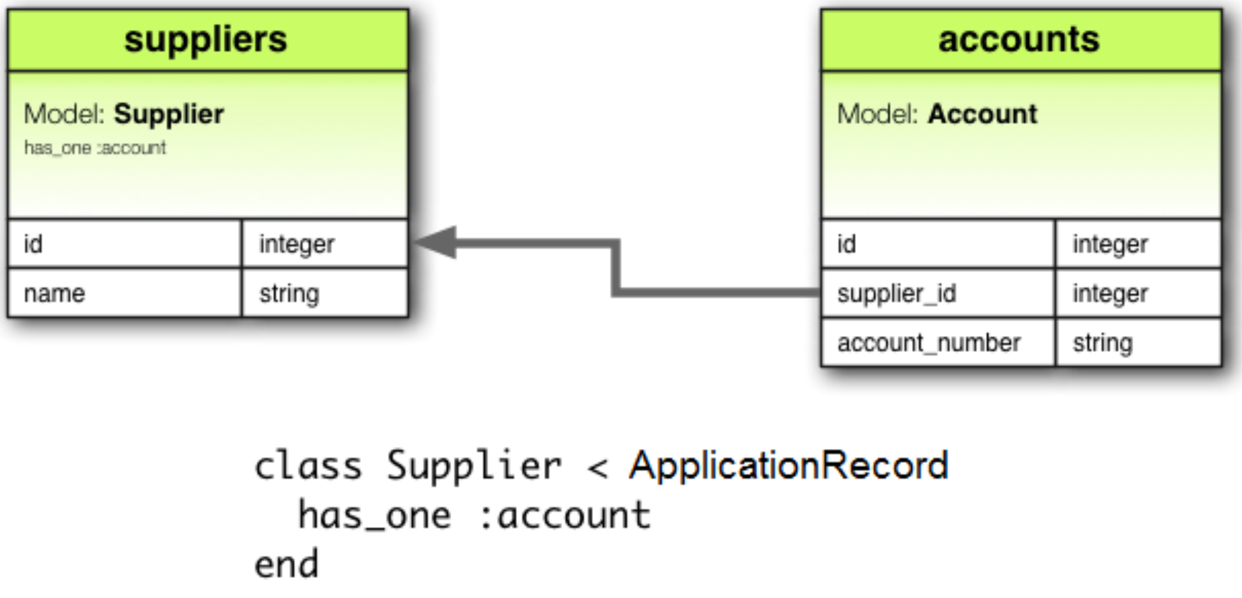
The corresponding migration might look like this:

```
class CreateBooks < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    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 < ApplicationRecord
  has_one :account
end
```



```
class Supplier < ApplicationRecord
  has_one :account
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

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

Depending on the use case, you might also need to create a unique index and/or a foreign key constraint on the supplier column for the accounts table. In this case, the column definition might look like this:

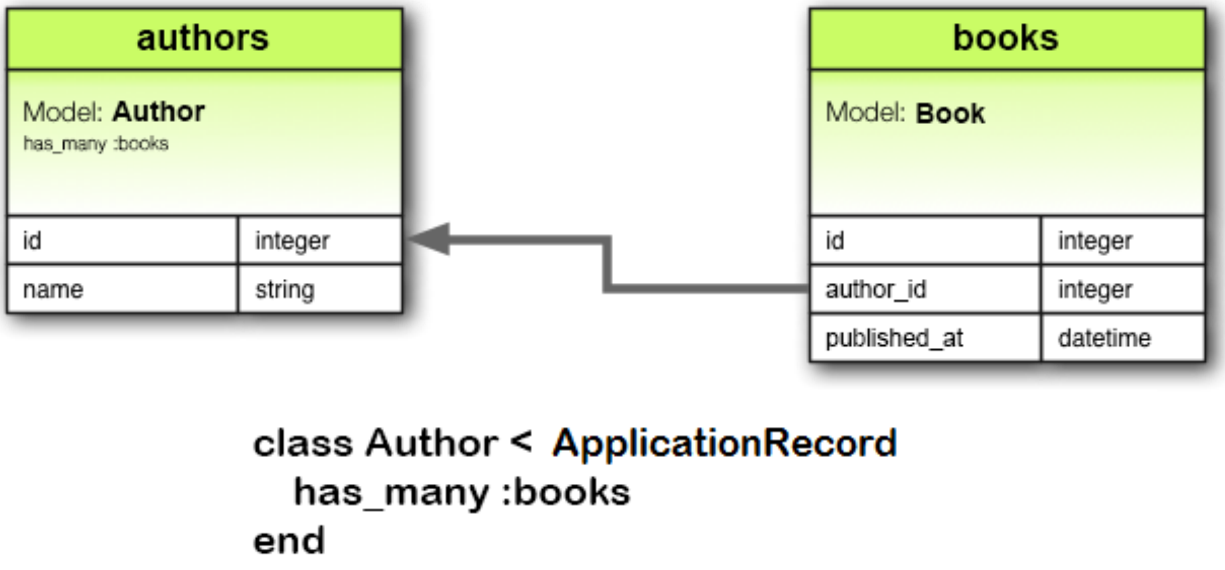
```
create_table :accounts do |t|
  t.belongs_to :supplier, index: { unique: true },
foreign_key: true
  # ...
end
```

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 authors and books, the author model could be declared like this:

```
class Author < ApplicationRecord
  has_many :books
end
```

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



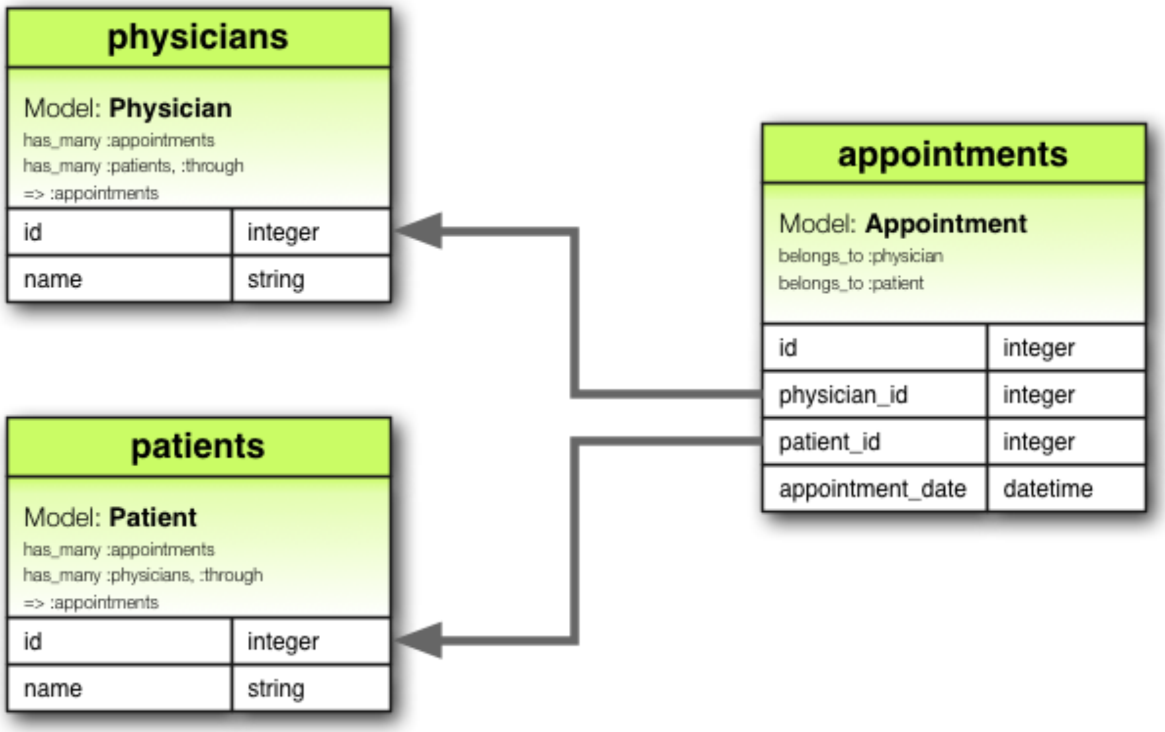
The corresponding migration might look like this:

```
class CreateAuthors < ActiveRecord::Migration[5.0]
  def change
    create_table :authors do |t|
      t.string :name
      t.timestamps
    end

    create_table :books do |t|
      t.belongs_to :author, index: true
      t.datetime :published_at
      t.timestamps
    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 < ApplicationRecord
  has_many :appointments
  has_many :patients, through: :appointments
end

class Appointment < ApplicationRecord
  belongs_to :physician
  belongs_to :patient
end

class Patient < ApplicationRecord
  has_many :appointments
  has_many :physicians, through: :appointments
end
```



```
class Physician < ApplicationRecord
  has_many :appointments
  has_many :patients, :through => :appointments
end

class Appointment < ApplicationRecord
  belongs_to :physician
  belongs_to :patient
end

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

The corresponding migration might look like this:

```
class CreateAppointments < ActiveRecord::Migration[5.0]
  def change
    create_table :physicians do |t|
      t.string :name
      t.timestamps
    end

    create_table :patients do |t|
      t.string :name
      t.timestamps
    end

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

The collection of join models can be managed via the has_many association methods. For example, if you assign:

```
physician.patients = patients
```

Then new join models are automatically created for the newly associated objects. If some that existed previously are now missing, then their join rows are automatically 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 < ApplicationRecord
  has_many :sections
  has_many :paragraphs, through: :sections
end

class Section < ApplicationRecord
  belongs_to :document
  has_many :paragraphs
end

class Paragraph < ApplicationRecord
  belongs_to :section
end
```

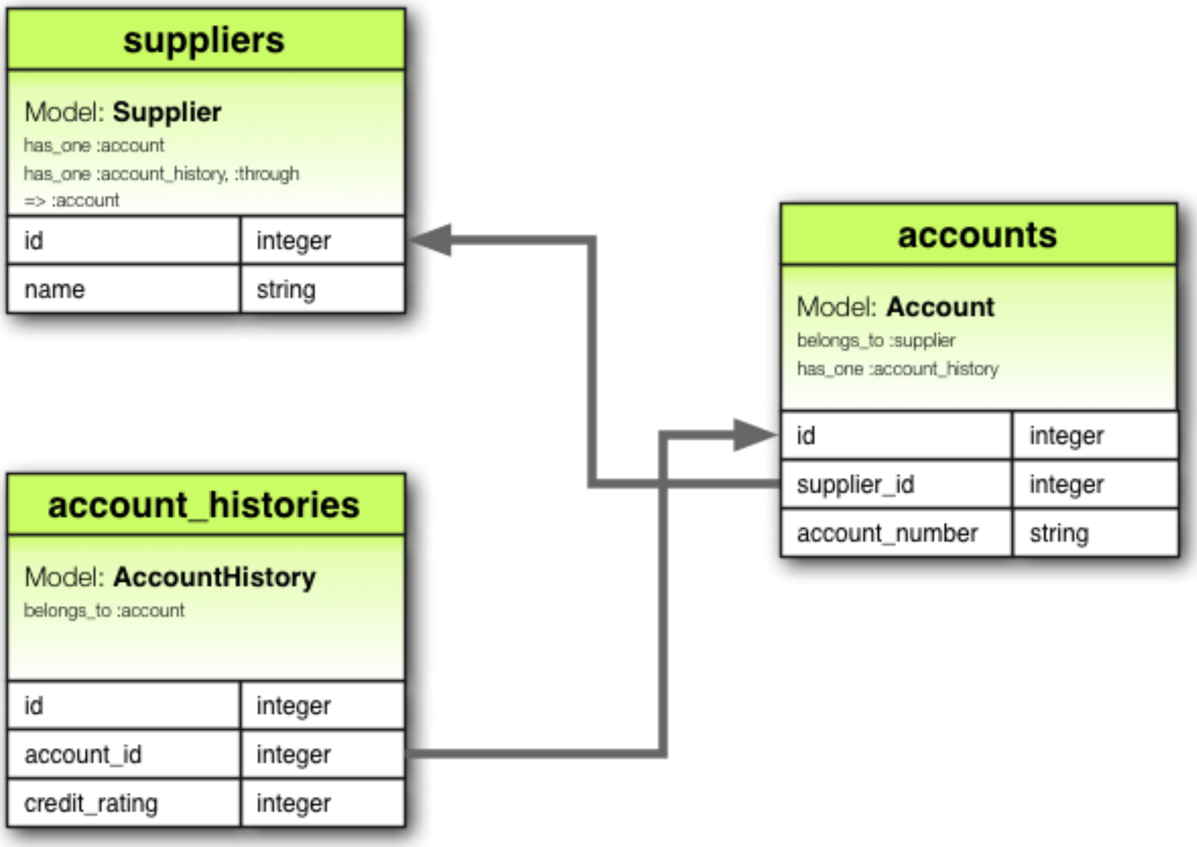
With `through: :sections` specified, Rails will now understand:

```
@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 < ApplicationRecord
  has_one :account
  has_one :account_history, through: :account
end

class Account < ApplicationRecord
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ApplicationRecord
  belongs_to :account
end
```



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

class Account < ApplicationRecord
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ApplicationRecord
  belongs_to :account
end
```

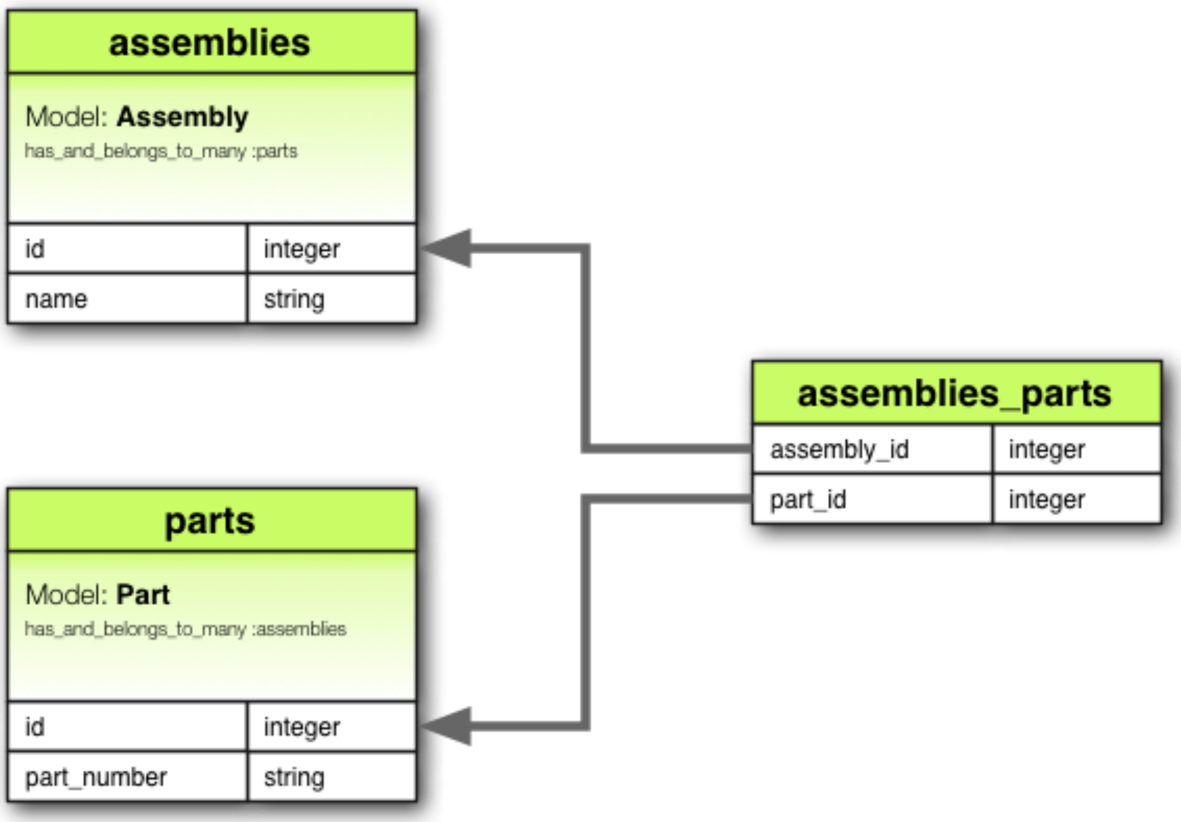
The corresponding migration might look like this:

```
class CreateAccountHistories <
ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

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

    create_table :account_histories do |t|
      t.belongs_to :account, index: true
      t.integer :credit_rating
      t.timestamps
    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 Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```



```
class Assembly < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  has_and_belongs_to_many :assemblies
end
```

The corresponding migration might look like this:

```
class CreateAssembliesAndParts <
ActiveRecord::Migration[5.0]
  def change
    create_table :assemblies do |t|
      t.string :name
      t.timestamps
    end

    create_table :parts do |t|
      t.string :part_number
      t.timestamps
    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 < ApplicationRecord
  has_one :account
end

class Account < ApplicationRecord
  belongs_to :supplier
end
```

The corresponding migration might look like this:

```
class CreateSuppliers < ActiveRecord::Migration[5.0]
  def change
    create_table :suppliers do |t|
      t.string :name
      t.timestamps
    end

    create_table :accounts do |t|
      t.integer :supplier_id
      t.string  :account_number
      t.timestamps
    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 < ApplicationRecord
  has_and_belongs_to_many :parts
end

class Part < ApplicationRecord
  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 < ApplicationRecord
  has_many :manifests
  has_many :parts, through: :manifests
end

class Manifest < ApplicationRecord
  belongs_to :assembly
  belongs_to :part
end

class Part < ApplicationRecord
  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 < ApplicationRecord
  belongs_to :imageable, polymorphic: true
end

class Employee < ApplicationRecord
  has_many :pictures, as: :imageable
end

class Product < ApplicationRecord
  has_many :pictures, as: :imageable
end
```

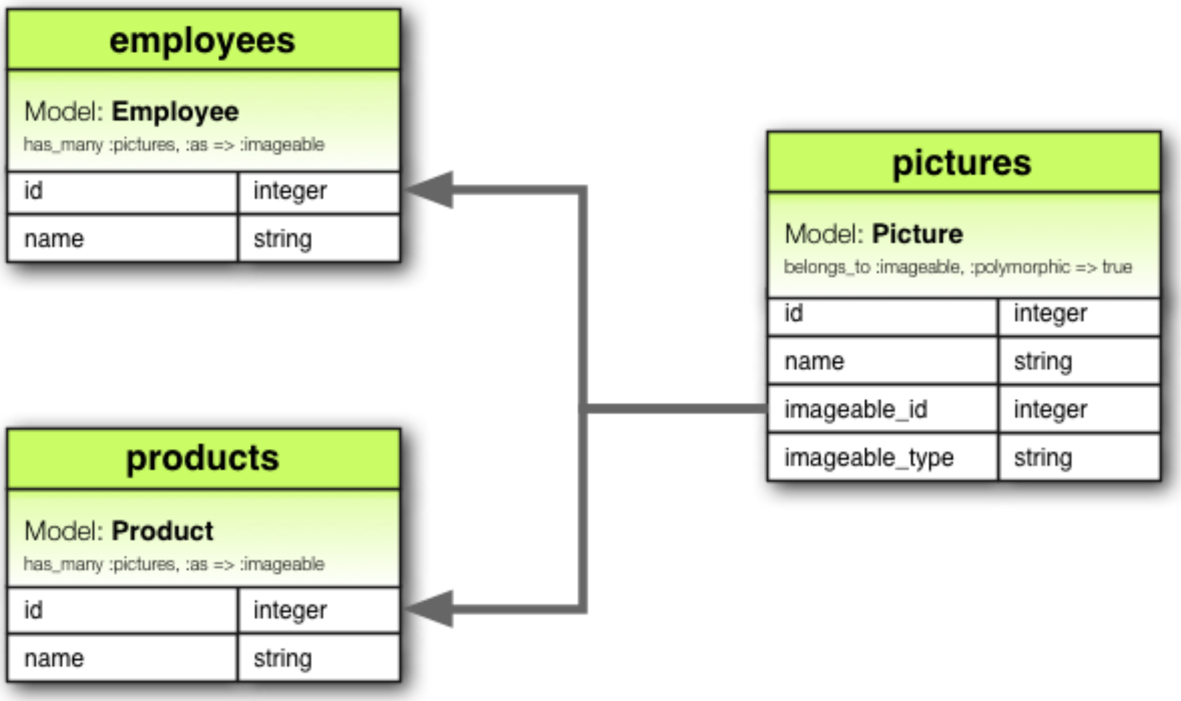
You can think of a polymorphic `belongs_to` declaration as setting up an interface that any other model can use. From an instance of the `Employee` model, you can retrieve a collection of pictures: `@employee.pictures`. Similarly, you can retrieve `@product.pictures`.

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[5.0]
  def change
    create_table :pictures do |t|
      t.string  :name
      t.integer :imageable_id
      t.string  :imageable_type
      t.timestamps
    end

    add_index :pictures, 
[:imageable_type, :imageable_id]
  end
end
```

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

```
class CreatePictures < ActiveRecord::Migration[5.0]
  def change
    create_table :pictures do |t|
      t.string :name
      t.references :imageable, polymorphic: true, index:
true
      t.timestamps
    end
  end
end
```



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

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

class Product < ApplicationRecord
  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:

```ruby
class Employee < ApplicationRecord
  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.

```ruby
class CreateEmployees < ActiveRecord::Migration[5.0]
  def change
    create_table :employees do |t|
      t.references :manager, index: true
      t.timestamps
    end
  end
end
```

Callbacks Introduction

1 The Object Life Cycle

During the normal operation of a Rails application, objects may be created, updated, and destroyed.

Active Record provides **hooks into this *object life cycle* so that you can control your application and its data.**

Callbacks allow you to trigger logic before or after an alteration of an object's state.

2 Callbacks Overview

Callbacks are methods that get called at certain moments of an object's life cycle. With callbacks it is possible to write code that will run whenever an Active Record object is created, saved, updated, deleted, validated, or loaded from the database.

2.1 Callback Registration

In order to use the available callbacks, you need to register them. You can implement the callbacks as ordinary methods and use a macro-style class method to register them as callbacks:

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_validation :ensure_login_has_a_value

  private
    def ensure_login_has_a_value
      if login.nil?
        self.login = email unless email.blank?
      end
    end
end
```

The macro-style class methods can also receive a block. Consider using this style if the code inside your block is so short that it fits in a single line:

```
class User < ApplicationRecord
  validates :login, :email, presence: true

  before_create do
    self.name = login.capitalize if name.blank?
  end
end
```

Callbacks can also be registered to only fire on certain life cycle events:

```
class User < ApplicationRecord
  before_validation :normalize_name, on: :create

  # :on takes an array as well
  after_validation :set_location, on: [ :create, :update
]

  private
    def normalize_name
      self.name = name.downcase.titleize
    end

    def set_location
      self.location = LocationService.query(self)
    end
end
```

It is considered good practice to declare callback methods as private. If left public, they can be called from outside of the model and violate the principle of object encapsulation.

3 Available Callbacks

Here is a list with all the available Active Record callbacks, listed in the same order in which they will get called during the respective operations:

3.1 Creating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_create`
- `around_create`
- `after_create`
- `after_save`
- `after_commit/after_rollback`

3.2 Updating an Object

- `before_validation`
- `after_validation`
- `before_save`
- `around_save`
- `before_update`
- `around_update`
- `after_update`
- `after_save`
- `after_commit/after_rollback`

3.3 Destroying an Object

- `before_destroy`
- `around_destroy`
- `after_destroy`
- `after_commit/after_rollback`

`after_save` runs both on create and update, but always *after* the more specific callbacks `after_create` and `after_update`, no matter the order in which the macro calls were executed.

`before_destroy` callbacks should be placed before `dependent: :destroy` associations (or use the `prepend: true` option), to ensure they execute before the records are deleted by `dependent: :destroy`.

3.4 `after_initialize` and `after_find`

The `after_initialize` callback will be called whenever an Active Record object is instantiated, either by directly using `new` or when a record is loaded from the database. It can be useful to avoid the need to directly override your Active Record `initialize` method.

The `after_find` callback will be called whenever Active Record loads a record from the database. `after_find` is called before `after_initialize` if both are defined.

The `after_initialize` and `after_find` callbacks have no `before_*` counterparts, but they can be registered just like the other Active Record callbacks.

```
class User < ApplicationRecord
  after_initialize do |user|
    puts "You have initialized an object!"
  end

  after_find do |user|
    puts "You have found an object!"
  end
end

>> User.new
You have initialized an object!
=> #<User id: nil>

>> User.first
You have found an object!
You have initialized an object!
=> #<User id: 1>
```

3.5 `after_touch`

The `after_touch` callback will be called whenever an Active Record object is touched.

```
class User < ApplicationRecord
  after_touch do |user|
    puts "You have touched an object"
  end
end

>> u = User.create(name: 'Kuldeep')
=> #<User id: 1, name: "Kuldeep", created_at:
"2013-11-25 12:17:49", updated_at: "2013-11-25
12:17:49">

>> u.touch
You have touched an object
=> true
```

It can be used along with `belongs_to`:

```
class Employee < ApplicationRecord
  belongs_to :company, touch: true
  after_touch do
    puts 'An Employee was touched'
  end
end

class Company < ApplicationRecord
  has_many :employees
  after_touch :log_when_employees_or_company_touched

  private
  def log_when_employees_or_company_touched
    puts 'Employee/Company was touched'
  end
end

>> @employee = Employee.last
=> #<Employee id: 1, company_id: 1, created_at:
"2013-11-25 17:04:22", updated_at: "2013-11-25
17:05:05">

# triggers @employee.company.touch
>> @employee.touch
Employee/Company was touched
An Employee was touched
=> true
```

4 Running Callbacks

The following methods trigger callbacks:

- `create`
- `create!`
- `destroy`
- `destroy!`
- `destroy_all`
- `save`
- `save!`
- `save(validate: false)`
- `toggle!`
- `touch`
- `update_attribute`
- `update`
- `update!`
- `valid?`

Additionally, the `after_find` callback is triggered by the following finder methods:

- `all`
- `first`
- `find`
- `find_by`
- `find_by_*`
- `find_by_*!`
- `find_by_sql`
- `last`

The `after_initialize` callback is triggered every time a new object of the class is initialized.

The `find_by_*` and `find_by_*!` methods are dynamic finders generated automatically for every attribute. Learn more about them at the Dynamic finders section

5 Skipping Callbacks

Just as with validations, it is also possible to skip callbacks by using the following methods:

- `decrement`
- `decrement_counter`
- `delete`
- `delete_all`
- `increment`
- `increment_counter`
- `toggle`
- `update_column`
- `update_columns`
- `update_all`
- `update_counters`

These methods should be used with caution, however, because important business rules and application logic may be kept in callbacks. Bypassing them without understanding the potential implications may lead to invalid data.

6 Halting Execution

As you start registering new callbacks for your models, they will be queued for execution. This queue will include all your model's validations, the registered callbacks, and the database operation to be executed.

The whole callback chain is wrapped in a transaction. If any callback raises an exception, the execution chain gets halted and a ROLLBACK is issued. To intentionally stop a chain use:

```
throw :abort
```

Any exception that is not `ActiveRecord::Rollback` or `ActiveRecord::RecordInvalid` will be re-raised by Rails after the callback chain is halted. Raising an exception other than `ActiveRecord::Rollback` or `ActiveRecord::RecordInvalid` may break code that does not expect methods like `save` and `update_attributes` (which normally try to return `true` or `false`) to raise an exception.

7 Relational Callbacks

Callbacks work through model relationships, and can even be defined by them. Suppose an example where a user has many articles. A user's articles should be destroyed if the user is destroyed. Let's add an `after_destroy` callback to the `User` model by way of its relationship to the `Article` model:

```
class User < ApplicationRecord
  has_many :articles, dependent: :destroy
end

class Article < ApplicationRecord
  after_destroy :log_destroy_action

  def log_destroy_action
    puts 'Article destroyed'
  end
end

>> user = User.first
=> #<User id: 1>
>> user.articles.create!
=> #<Article id: 1, user_id: 1>
>> user.destroy
Article destroyed
=> #<User id: 1>
```

8 Conditional Callbacks

As with validations, we can also make the calling of a callback method conditional on the satisfaction of a given predicate. We can do this using the `:if` and `:unless` options, which can take a symbol, a `Proc` or an `Array`. You may use the `:if` option when you want to specify under which conditions the callback **should** be called. If you want to specify the conditions under which the callback **should not** be called, then you may use the `:unless` option.

8.1 Using `:if` and `:unless` with a `Symbol`

You can associate the `:if` and `:unless` options with a symbol corresponding to the name of a predicate method that will get called right before the callback. When using the `:if` option, the callback won't be executed if the predicate method returns false; when using the `:unless` option, the callback won't be executed if the predicate method returns true. This is the most common option. Using this form of registration it is also possible to register several different predicates that should be called to check if the callback should be executed.

```
class Order < ApplicationRecord
  before_save :normalize_card_number, 
if: :paid_with_card?
end
```

8.2 Using `:if` and `:unless` with a `Proc`

Finally, it is possible to associate `:if` and `:unless` with a `Proc` object. This option is best suited when writing short validation methods, usually one-liners:

```
class Order < ApplicationRecord
  before_save :normalize_card_number,
    if: Proc.new { |order| order.paid_with_card? }
end
```

8.3 Multiple Conditions for Callbacks

When writing conditional callbacks, it is possible to mix both `:if` and `:unless` in the same callback declaration:

```
class Comment < ApplicationRecord
  after_create :send_email_to_author,
if: :author_wants_emails?,
    unless: Proc.new { |comment|
comment.article.ignore_comments? }
end
```

9 Callback Classes

Sometimes the callback methods that you'll write will be useful enough to be reused by other models. Active Record makes it possible to create classes that encapsulate the callback methods, so it becomes very easy to reuse them.
Here's an example where we create a class with an `after_destroy` callback for a `PictureFile` model:

```
class PictureFileCallbacks
  def after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

When declared inside a class, as above, the callback methods will receive the model object as a parameter. We can now use the callback class in the model:

```
class PictureFile < ApplicationRecord
  after_destroy PictureFileCallbacks.new
end
```

Note that we needed to instantiate a new `PictureFileCallbacks` object, since we declared our callback as an instance method. This is particularly useful if the callbacks make use of the state of the instantiated object. Often, however, it will make more sense to declare the callbacks as class methods:

```
class PictureFileCallbacks
  def self.after_destroy(picture_file)
    if File.exist?(picture_file.filepath)
      File.delete(picture_file.filepath)
    end
  end
end
```

If the callback method is declared this way, it won't be necessary to instantiate a `PictureFileCallbacks` object.

```
class PictureFile < ApplicationRecord
  after_destroy PictureFileCallbacks
end
```

You can declare as many callbacks as you want inside your callback classes.

10 Transaction Callbacks

There are two additional callbacks that are triggered by the completion of a database transaction: `after_commit` and `after_rollback`. These callbacks are very similar to the `after_save` callback except that they don't execute until after database changes have either been committed or rolled back.
They are most useful when your active record models need to interact with external systems which are not part of the database transaction.

Consider, for example, the previous example where the `PictureFile` model needs to delete a file after the corresponding record is destroyed. If anything raises an exception after the `after_destroy` callback is called and the transaction rolls back, the file will have been deleted and the model will be left in an inconsistent state. For example, suppose that `picture_file_2` in the code below is not valid and the `save!` method raises an error.

```
PictureFile.transaction do
  picture_file_1.destroy
  picture_file_2.save!
end
```

By using the `after_commit` callback we can account for this case.

```
class PictureFile < ApplicationRecord
  after_commit :delete_picture_file_from_disk, 
on: :destroy

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

The `:on` option specifies when a callback will be fired. If you don't supply the `:on` option the callback will fire for every action.

Since using `after_commit` callback only on create, update or delete is common, there are aliases for those operations:

- `after_create_commit`
- `after_update_commit`
- `after_destroy_commit`

```
class PictureFile < ApplicationRecord
  after_destroy_commit :delete_picture_file_from_disk

  def delete_picture_file_from_disk
    if File.exist?(filepath)
      File.delete(filepath)
    end
  end
end
```

The `after_commit` and `after_rollback` callbacks are called for all models created, updated, or destroyed within a transaction block. However, if an exception is raised within one of these callbacks, the exception will bubble up and any remaining `after_commit` or `after_rollback` methods will *not* be executed. As such, if your callback code could raise an exception, you'll need to rescue it and handle it within the callback in order to allow other callbacks to run.

Using both `after_create_commit` and `after_update_commit` in the same model will only allow the last callback defined to take effect, and will override all others.

```ruby
class User < ApplicationRecord
  after_create_commit :log_user_saved_to_db
  after_update_commit :log_user_saved_to_db

  private
  def log_user_saved_to_db
    puts 'User was saved to database'
  end
end

# prints nothing
>> @user = User.create

# updating @user
>> @user.save
=> User was saved to database
```

To register callbacks for both create and update actions, use `after_commit` instead.

```ruby
class User < ApplicationRecord
  after_commit :log_user_saved_to_db, on: 
[:create, :update]
end
```

1 Validations Overview

Here's an example of a very simple validation:

```ruby
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

As you can see, our validation lets us know that our `Person` is not valid without a `name` attribute. The second `Person` will not be persisted to the database.

Before we dig into more details, let's talk about how validations fit into the big picture of your application.

1.1 Why Use Validations?

Validations are used to ensure that only valid data is saved into your database. For example, it may be important to your application to ensure that every user provides a valid email address and mailing address.

Model-level validations are the best way to ensure that only valid data is saved into your database. They are database agnostic, cannot be bypassed by end users, and are convenient to test and maintain. Rails makes them easy to use, provides built-in helpers for common needs, and allows you to create your own validation methods as well.

There are several other ways to validate data before it is saved into your database, including native database constraints, client-side validations and controller-level validations. Here's a summary of the pros and cons:

- Database constraints and/or stored procedures make the validation mechanisms database-dependent and can make testing and maintenance more difficult. However, if your database is used by other applications, it may be a good idea to use some constraints at the database level. Additionally, database-level validations can safely handle some things (such as uniqueness in heavily-used tables) that can be difficult to implement otherwise.
- Client-side validations can be useful, but are generally unreliable if used alone. If they are implemented using JavaScript, they may be bypassed if JavaScript is turned off in the user's browser. However, if combined with other techniques, client-side validation can be a convenient way to provide users with immediate feedback as they use your site.
- Controller-level validations can be tempting to use, but often become unwieldy and difficult to test and maintain. Whenever possible, it's a good idea to keep your controllers skinny, as it will make your application a pleasure to work with in the long run.

Choose these in certain, specific cases. It's the opinion of the Rails team that model-level validations are the most appropriate in most circumstances.

1.2 When Does Validation Happen?

There are two kinds of Active Record objects: those that correspond to a row inside your database and those that do not. When you create a fresh object, for example using the `new` method, that object does not belong to the database yet. Once you call `save` upon that object it will be saved into the appropriate database table. Active Record uses the `new_record?` instance method to determine whether an object is already in the database or not. Consider the following simple Active Record class:

```
class Person < ApplicationRecord
end
```

We can see how it works by looking at some `rails console` output:

```
$ bin/rails console
>> p = Person.new(name: "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, 
updated_at: nil>
>> p.new_record?
=> true
>> p.save
=> true
>> p.new_record?
=> false
```

Creating and saving a new record will send an SQL `INSERT` operation to the database. Updating an existing record will send an SQL `UPDATE` operation instead. Validations are typically run before these commands are sent to the database. If any validations fail, the object will be marked as invalid and Active Record will not perform the `INSERT` or `UPDATE` operation. This avoids storing an invalid object in the database. You can choose to have specific validations run when an object is created, saved, or updated.

There are many ways to change the state of an object in the database. Some methods will trigger validations, but some will not. This means that it's possible to save an object in the database in an invalid state if you aren't careful.

The following methods trigger validations, and will save the object to the database only if the object is valid:

- `create`
- `create!`
- `save`
- `save!`
- `update`
- `update!`

The bang versions (e.g. `save!`) raise an exception if the record is invalid. The non-bang versions don't: `save` and `update` return `false`, and `create` just returns the object.

1.3 Skipping Validations

The following methods skip validations, and will save the object to the database regardless of its validity. They should be used with caution.

- `decrement!`
- `decrement_counter`
- `increment!`
- `increment_counter`
- `toggle!`
- `touch`
- `update_all`
- `update_attribute`
- `update_column`
- `update_columns`
- `update_counters`

Note that `save` also has the ability to skip validations if passed `validate: false` as an argument. This technique should be used with caution.

- `save(validate: false)`

1.4 `valid?` and `invalid?`

Before saving an Active Record object, Rails runs your validations. If these validations produce any errors, Rails does not save the object.

You can also run these validations on your own. `valid?` triggers your validations and returns true if no errors were found in the object, and false otherwise. As you saw above:

```
class Person < ApplicationRecord
  validates :name, presence: true
end

Person.create(name: "John Doe").valid? # => true
Person.create(name: nil).valid? # => false
```

After Active Record has performed validations, any errors found can be accessed through the `errors.messages` instance method, which returns a collection of errors. By definition, an object is valid if this collection is empty after running validations.
Note that an object instantiated with `new` will not report errors even if it's technically invalid, because validations are automatically run only when the object is saved, such as with the `create` or `save` methods.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> p = Person.new
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {}

>> p.valid?
# => false
>> p.errors.messages
# => {name:["can't be blank"]}

>> p = Person.create
# => #<Person id: nil, name: nil>
>> p.errors.messages
# => {name:["can't be blank"]}

>> p.save
# => false

>> p.save!
# => ActiveRecord::RecordInvalid: Validation failed: 
Name can't be blank

>> Person.create!
# => ActiveRecord::RecordInvalid: Validation failed: 
Name can't be blank
```

`invalid?` is simply the inverse of `valid?`. It triggers your validations, returning true if any errors were found in the object, and false otherwise.

1.5 `errors[]`

To verify whether or not a particular attribute of an object is valid, you can use `errors[:attribute]`. It returns an array of all the errors for `:attribute`. If there are no errors on the specified attribute, an empty array is returned.

This method is only useful *after* validations have been run, because it only inspects the errors collection and does not trigger validations itself. It's different from the `ActiveRecord::Base#invalid?` method explained above because it doesn't verify the validity of the object as a whole. It only checks to see whether there are errors found on an individual attribute of the object.

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> Person.new.errors[:name].any? # => false
>> Person.create.errors[:name].any? # => true
```

We'll cover validation errors in greater depth in the Working with Validation Errors section.

1.6 `errors.details`

To check which validations failed on an invalid attribute, you can use `errors.details[:attribute]`. It returns an array of hashes with an `:error` key to get the symbol of the validator:

```
class Person < ApplicationRecord
  validates :name, presence: true
end

>> person = Person.new
>> person.valid?
>> person.errors.details[:name] # => [{error: :blank}]
```

Using `details` with custom validators is covered in the Working with Validation Errors section.

2 Validation Helpers

Active Record offers many pre-defined validation helpers that you can use directly inside your class definitions. These helpers provide common validation rules. Every time a validation fails, an error message is added to the object's `errors` collection, and this message is associated with the attribute being validated.

Each helper accepts an arbitrary number of attribute names, so with a single line of code you can add the same kind of validation to several attributes.
All of them accept the `:on` and `:message` options, which define when the validation should be run and what message should be added to the `errors` collection if it fails, respectively. The `:on` option takes one of the values `:create` or `:update`. There is a default error message for each one of the validation helpers. These messages are used when the `:message` option isn't specified. Let's take a look at each one of the available helpers.

2.1 `acceptance`

This method validates that a checkbox on the user interface was checked when a form was submitted. This is typically used when the user needs to agree to your application's terms of service, confirm that some text is read, or any similar concept.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: true
end
```

This check is performed only if `terms_of_service` is not `nil`. The default error message for this helper is *"must be accepted"*. You can also pass custom message via the `message` option.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: { message: 
'must be abided' }
end
```

It can also receive an `:accept` option, which determines the allowed values that will be considered as accepted. It defaults to `['1', true]` and can be easily changed.

```
class Person < ApplicationRecord
  validates :terms_of_service, acceptance: { accept: 
'yes' }
  validates :eula, acceptance: { accept: ['TRUE', 
'accepted'] }
end
```

This validation is very specific to web applications and this 'acceptance' does not need to be recorded anywhere in your database. If you don't have a field for it, the helper will just create a virtual attribute. If the field does exist in your database, the `accept` option must be set to or include `true` or else the validation will not run.

2.2 `validates_associated`

You should use this helper when your model has associations with other models and they also need to be validated. When you try to save your object, `valid?` will be called upon each one of the associated objects.

```
class Library < ApplicationRecord
  has_many :books
  validates_associated :books
end
```

This validation will work with all of the association types.

Don't use `validates_associated` on both ends of your associations. They would call each other in an infinite loop.

The default error message for `validates_associated` is *"is invalid"*. Note that each associated object will contain its own `errors` collection; errors do not bubble up to the calling model.

2.3 `confirmation`

You should use this helper when you have two text fields that should receive exactly the same content. For example, you may want to confirm an email address or a password. This validation creates a virtual attribute whose name is the name of the field that has to be confirmed with "_confirmation" appended.

```
class Person < ApplicationRecord
  validates :email, confirmation: true
end
```

In your view template you could use something like

```
<%= text_field :person, :email %>
<%= text_field :person, :email_confirmation %>
```

This check is performed only if `email_confirmation` is not `nil`. To require confirmation, make sure to add a presence check for the confirmation attribute (we'll take a look at `presence` later on in this guide):

```
class Person < ApplicationRecord
  validates :email, confirmation: true
  validates :email_confirmation, presence: true
end
```

There is also a `:case_sensitive` option that you can use to define whether the confirmation constraint will be case sensitive or not. This option defaults to true.

```
class Person < ApplicationRecord
  validates :email, confirmation: { case_sensitive: 
false }
end
```

The default error message for this helper is *"doesn't match confirmation"*.

2.4 `exclusion`

This helper validates that the attributes' values are not included in a given set. In fact, this set can be any enumerable object.

```
class Account < ApplicationRecord
  validates :subdomain, exclusion: { in: %w(www us ca 
jp),
    message: "%{value} is reserved." }
end
```

The `exclusion` helper has an option `:in` that receives the set of values that will not be accepted for the validated attributes. The `:in` option has an alias called `:within` that you can use for the same purpose, if you'd like to. This example uses the `:message` option to show how you can include the attribute's value. For full options to the message argument please see the message documentation.

The default error message is *"is reserved"*.

2.5 `format`

This helper validates the attributes' values by testing whether they match a given regular expression, which is specified using the `:with` option.

```
class Product < ApplicationRecord
  validates :legacy_code, format: { with: /\A[a-zA-Z]+
\z/,
    message: "only allows letters" }
end
```

Alternatively, you can require that the specified attribute does *not* match the regular expression by using the `:without` option.

The default error message is *"is invalid"*.

2.6 `inclusion`

This helper validates that the attributes' values are included in a given set. In fact, this set can be any enumerable object.

```
class Coffee < ApplicationRecord
  validates :size, inclusion: { in: %w(small medium 
large),
    message: "%{value} is not a valid size" }
end
```

The `inclusion` helper has an option `:in` that receives the set of values that will be accepted. The `:in` option has an alias called `:within` that you can use for the same purpose, if you'd like to. The previous example uses the `:message` option to show how you can include the attribute's value. For full options please see the message documentation.

The default error message for this helper is *"is not included in the list"*.

2.7 `length`

This helper validates the length of the attributes' values. It provides a variety of options, so you can specify length constraints in different ways:

```
class Person < ApplicationRecord
  validates :name, length: { minimum: 2 }
  validates :bio, length: { maximum: 500 }
  validates :password, length: { in: 6..20 }
  validates :registration_number, length: { is: 6 }
end
```

The possible length constraint options are:

- `:minimum` - The attribute cannot have less than the specified length.
- `:maximum` - The attribute cannot have more than the specified length.
- `:in` (or `:within`) - The attribute length must be included in a given interval. The value for this option must be a range.
- `:is` - The attribute length must be equal to the given value.

The default error messages depend on the type of length validation being performed. You can personalize these messages using the `:wrong_length`, `:too_long`, and `:too_short` options and `%{count}` as a placeholder for the number corresponding to the length constraint being used. You can still use the `:message` option to specify an error message.

```
class Person < ApplicationRecord
  validates :bio, length: { maximum: 1000,
    too_long: "%{count} characters is the maximum 
allowed" }
end
```

Note that the default error messages are plural (e.g., "is too short (minimum is %{count} characters)"). For this reason, when `:minimum` is 1 you should provide a personalized message or use `presence: true` instead. When `:in` or `:within` have a lower limit of 1, you should either provide a personalized message or call `presence` prior to `length`.

2.8 `numericality`

This helper validates that your attributes have only numeric values. By default, it will match an optional sign followed by an integral or floating point number. To specify that only integral numbers are allowed set `:only_integer` to true.

If you set `:only_integer` to `true`, then it will use the

```
/\A[+-]?\d+\z/
```

regular expression to validate the attribute's value. Otherwise, it will try to convert the value to a number using `Float`.

```
class Player < ApplicationRecord
  validates :points, numericality: true
  validates :games_played, numericality: { only_integer: true }
end
```

Besides `:only_integer`, this helper also accepts the following options to add constraints to acceptable values:

- `:greater_than` - Specifies the value must be greater than the supplied value. The default error message for this option is *"must be greater than %{count}"*.
- `:greater_than_or_equal_to` - Specifies the value must be greater than or equal to the supplied value. The default error message for this option is *"must be greater than or equal to %{count}"*.
- `:equal_to` - Specifies the value must be equal to the supplied value. The default error message for this option is *"must be equal to %{count}"*.
- `:less_than` - Specifies the value must be less than the supplied value. The default error message for this option is *"must be less than %{count}"*.
- `:less_than_or_equal_to` - Specifies the value must be less than or equal to the supplied value. The default error message for this option is *"must be less than or equal to %{count}"*.
- `:other_than` - Specifies the value must be other than the supplied value. The default error message for this option is *"must be other than %{count}"*.
- `:odd` - Specifies the value must be an odd number if set to true. The default error message for this option is *"must be odd"*.
- `:even` - Specifies the value must be an even number if set to true. The default error message for this option is *"must be even"*.

By default, `numericality` doesn't allow `nil` values. You can use `allow_nil: true` option to permit it.

The default error message is *"is not a number"*.

2.9 `presence`

This helper validates that the specified attributes are not empty. It uses the `blank?` method to check if the value is either `nil` or a blank string, that is, a string that is either empty or consists of whitespace.

```
class Person < ApplicationRecord
  validates :name, :login, :email, presence: true
end
```

If you want to be sure that an association is present, you'll need to test whether the associated object itself is present, and not the foreign key used to map the association.

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, presence: true
end
```

In order to validate associated records whose presence is required, you must specify the `:inverse_of` option for the association:

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

If you validate the presence of an object associated via a `has_one` or `has_many` relationship, it will check that the object is neither `blank?` nor `marked_for_destruction?`.

Since `false.blank?` is true, if you want to validate the presence of a boolean field you should use one of the following validations:

```
validates :boolean_field_name, inclusion: { in: [true, false] }
validates :boolean_field_name, exclusion: { in: [nil] }
```

By using one of these validations, you will ensure the value will NOT be `nil` which would result in a `NULL` value in most cases.

2.10 `absence`

This helper validates that the specified attributes are absent. It uses the `present?` method to check if the value is not either nil or a blank string, that is, a string that is either empty or consists of whitespace.

```
class Person < ApplicationRecord
  validates :name, :login, :email, absence: true
end
```

If you want to be sure that an association is absent, you'll need to test whether the associated object itself is absent, and not the foreign key used to map the association.

```
class LineItem < ApplicationRecord
  belongs_to :order
  validates :order, absence: true
end
```

In order to validate associated records whose absence is required, you must specify the `:inverse_of` option for the association:

```
class Order < ApplicationRecord
  has_many :line_items, inverse_of: :order
end
```

If you validate the absence of an object associated via a `has_one` or `has_many` relationship, it will check that the object is neither `present?` nor `marked_for_destruction?`.

Since `false.present?` is false, if you want to validate the absence of a boolean field you should use `validates :field_name, exclusion: { in: [true, false] }`.

The default error message is *"must be blank"*.

2.11 `uniqueness`

This helper validates that the attribute's value is unique right before the object gets saved. It does not create a uniqueness constraint in the database, so it may happen that two different database connections create two records with the same value for a column that you intend to be unique. To avoid that, you must create a unique index on that column in your database.

```
class Account < ApplicationRecord
  validates :email, uniqueness: true
end
```

The validation happens by performing an SQL query into the model's table, searching for an existing record with the same value in that attribute.

There is a `:scope` option that you can use to specify one or more attributes that are used to limit the uniqueness check:

```
class Holiday < ApplicationRecord
  validates :name, uniqueness: { scope: :year,
    message: "should happen once per year" }
end
```

Should you wish to create a database constraint to prevent possible violations of a uniqueness validation using the `:scope` option, you must create a unique index on both columns in your database. See the MySQL manual for more details about multiple column indexes or the PostgreSQL manual for examples of unique constraints that refer to a group of columns.

There is also a `:case_sensitive` option that you can use to define whether the uniqueness constraint will be case sensitive or not. This option defaults to true.

```
class Person < ApplicationRecord
  validates :name, uniqueness: { case_sensitive: false }
end
```

Note that some databases are configured to perform case-insensitive searches anyway.
The default error message is *"has already been taken"*.

2.12 `validates_with`

This helper passes the record to a separate class for validation.

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if record.first_name == "Evil"
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator
end
```

Errors added to `record.errors[:base]` relate to the state of the record as a whole, and not to a specific attribute.

The `validates_with` helper takes a class, or a list of classes to use for validation. There is no default error message for `validates_with`. You must manually add errors to the record's errors collection in the validator class.

To implement the validate method, you must have a `record` parameter defined, which is the record to be validated.
Like all other validations, `validates_with` takes the `:if`, `:unless` and `:on` options. If you pass any other options, it will send those options to the validator class as `options`:

```
class GoodnessValidator < ActiveModel::Validator
  def validate(record)
    if options[:fields].any?{|field| record.send(field)
== "Evil" }
      record.errors[:base] << "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator, fields:
[:first_name, :last_name]
end
```

Note that the validator will be initialized *only once* for the whole application life cycle, and not on each validation run, so be careful about using instance variables inside it.
If your validator is complex enough that you want instance variables, you can easily use a plain old Ruby object instead:

```
class Person < ApplicationRecord
  validate do |person|
    GoodnessValidator.new(person).validate
  end
end

class GoodnessValidator
  def initialize(person)
    @person = person
  end

  def validate
    if
some_complex_condition_involving_ivars_and_private_metho
ds?
      @person.errors[:base] << "This person is evil"
    end
  end

  # ...
end
```

2.13 `validates_each`

This helper validates attributes against a block. It doesn't have a predefined validation function. You should create one using a block, and every attribute passed to `validates_each` will be tested against it. In the following example, we don't want names and surnames to begin with lower case.

```
class Person < ApplicationRecord
  validates_each :name, :surname do |record, attr,
value|
    record.errors.add(attr, 'must start with upper
case') if value =~ /\A[[:lower:]]/
  end
end
```

The block receives the record, the attribute's name and the attribute's value. You can do anything you like to check for valid data within the block. If your validation fails, you should add an error message to the model, therefore making it invalid.

3 Common Validation Options

These are common validation options:

3.1 `:allow_nil`

The `:allow_nil` option skips the validation when the value being validated is `nil`.

```
class Coffee < ApplicationRecord
  validates :size, inclusion: { in: %w(small medium
large),
    message: "%{value} is not a valid size" },
allow_nil: true
end
```

For full options to the message argument please see the message documentation.

3.2 `:allow_blank`

The `:allow_blank` option is similar to the `:allow_nil` option. This option will let validation pass if the attribute's value is `blank?`, like `nil` or an empty string for example.

```
class Topic < ApplicationRecord
  validates :title, length: { is: 5 }, allow_blank: true
end

Topic.create(title: "").valid?  # => true
Topic.create(title: nil).valid? # => true
```

3.3 `:message`

As you've already seen, the `:message` option lets you specify the message that will be added to the `errors` collection when validation fails. When this option is not used, Active Record will use the respective default error message for each validation helper. The `:message` option accepts a `String` or `Proc`.

A `String` `:message` value can optionally contain any/all of `%{value}`, `%{attribute}`, and `%{model}` which will be dynamically replaced when validation fails. This replacement is done using the I18n gem, and the placeholders must match exactly, no spaces are allowed.

A `Proc` `:message` value is given two arguments: the object being validated, and a hash with `:model`, `:attribute`, and `:value` key-value pairs.

```
class Person < ApplicationRecord
  # Hard-coded message
  validates :name, presence: { message: "must be given 
please" }

  # Message with dynamic attribute value. %{value} will 
be replaced with
  # the actual value of the attribute. %{attribute} and 
%{model} also
  # available.
  validates :age, numericality: { message: "%{value} 
seems wrong" }

  # Proc
  validates :username,
    uniqueness: {
      # object = person object being validated
      # data = { model: "Person", attribute: "Username", 
value: <username> }
      message: ->(object, data) do
        "Hey #{object.name}!, #{data[:value]} is taken 
already! Try again #{Time.zone.tomorrow}"
      end
    }
end
```

3.4 `:on`

The `:on` option lets you specify when the validation should happen. The default behavior for all the built-in validation helpers is to be run on save (both when you're creating a new record and when you're updating it). If you want to change it, you can use `on: :create` to run the validation only when a new record is created or `on: :update` to run the validation only when a record is updated.

```
class Person < ApplicationRecord
  # it will be possible to update email with a 
duplicated value
  validates :email, uniqueness: true, on: :create

  # it will be possible to create the record with a non-
numerical age
  validates :age, numericality: true, on: :update

  # the default (validates on both create and update)
  validates :name, presence: true
end
```

You can also use `on:` to define custom context. Custom contexts need to be triggered explicitly by passing name of the context to `valid?`, `invalid?` or `save`.

```
class Person < ApplicationRecord
  validates :email, uniqueness: true, on: :account_setup
  validates :age, numericality: true, on: :account_setup
end

person = Person.new
```

`person.valid?(:account_setup)` executes both the validations without saving the model. And `person.save(context: :account_setup)` validates `person` in `account_setup` context before saving. On explicit triggers, model is validated by validations of only that context and validations without context.

4 Strict Validations

You can also specify validations to be strict and raise `ActiveModel::StrictValidationFailed` when the object is invalid.

```
class Person < ApplicationRecord
  validates :name, presence: { strict: true }
end

Person.new.valid?  # => 
ActiveModel::StrictValidationFailed: Name can't be blank
```

There is also the ability to pass a custom exception to the `:strict` option.

```
class Person < ApplicationRecord
  validates :token, presence: true, uniqueness: true, 
strict: TokenGenerationException
end

Person.new.valid?  # => TokenGenerationException: Token 
can't be blank
```

5 Conditional Validation

Sometimes it will make sense to validate an object only when a given predicate is satisfied. You can do that by using the `:if` and `:unless` options, which can take a symbol, a `Proc` or an `Array`. You may use the `:if` option when you want to specify when the validation **should** happen. If you want to specify when the validation **should not** happen, then you may use the `:unless` option.

5.1 Using a Symbol with `:if` and `:unless`

You can associate the `:if` and `:unless` options with a symbol corresponding to the name of a method that will get called right before validation happens. This is the most commonly used option.

```
class Order < ApplicationRecord
  validates :card_number, presence: true, 
if: :paid_with_card?

  def paid_with_card?
    payment_type == "card"
  end
end
```

5.2 Using a Proc with `:if` and `:unless`

Finally, it's possible to associate `:if` and `:unless` with a `Proc` object which will be called. Using a `Proc` object gives you the ability to write an inline condition instead of a separate method. This option is best suited for one-liners.

```
class Account < ApplicationRecord
  validates :password, confirmation: true,
    unless: Proc.new { |a| a.password.blank? }
end
```

5.3 Grouping Conditional validations

Sometimes it is useful to have multiple validations use one condition. It can be easily achieved using `with_options`.

```
class User < ApplicationRecord
  with_options if: :is_admin? do |admin|
    admin.validates :password, length: { minimum: 10 }
    admin.validates :email, presence: true
  end
end
```

All validations inside of the `with_options` block will have automatically passed the condition `if: :is_admin?`

5.4 Combining Validation Conditions

On the other hand, when multiple conditions define whether or not a validation should happen, an `Array` can be used. Moreover, you can apply both `:if` and `:unless` to the same validation.

```
class Computer < ApplicationRecord
  validates :mouse, presence: true,
                    if: [Proc.new { |c| c.market.retail? 
}, :desktop?],
                    unless: Proc.new { |c| 
c.trackpad.present? }
end
```

The validation only runs when all the `:if` conditions and none of the `:unless` conditions are evaluated to `true`.

6 Performing Custom Validations

When the built-in validation helpers are not enough for your needs, you can write your own validators or validation methods as you prefer.

6.1 Custom Validators

Custom validators are classes that inherit from `ActiveModel::Validator`. These classes must implement the `validate` method which takes a record as an argument and performs the validation on it. The custom validator is called using the `validates_with` method.

```
class MyValidator < ActiveModel::Validator
  def validate(record)
    unless record.name.starts_with? 'X'
      record.errors[:name] << 'Need a name starting with 
X please!'
    end
  end
end

class Person
  include ActiveModel::Validations
  validates_with MyValidator
end
```

The easiest way to add custom validators for validating individual attributes is with the convenient `ActiveModel::EachValidator`. In this case, the custom validator class must implement a `validate_each` method which takes three arguments: record, attribute, and value. These correspond to the instance, the attribute to be validated, and the value of the attribute in the passed instance.

```
class EmailValidator < ActiveModel::EachValidator
  def validate_each(record, attribute, value)
    unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]
{2,})\z/i
      record.errors[attribute] << (options[:message] ||
"is not an email")
    end
  end
end

class Person < ApplicationRecord
  validates :email, presence: true, email: true
end
```

As shown in the example, you can also combine standard validations with your own custom validators.

6.2 Custom Methods

You can also create methods that verify the state of your models and add messages to the `errors` collection when they are invalid. You must then register these methods by using the `validate` (API) class method, passing in the symbols for the validation methods' names.
You can pass more than one symbol for each class method and the respective validations will be run in the same order as they were registered.
The `valid?` method will verify that the errors collection is empty, so your custom validation methods should add errors to it when you wish validation to fail:

```
class Invoice < ApplicationRecord
  validate :expiration_date_cannot_be_in_the_past,
    :discount_cannot_be_greater_than_total_value

  def expiration_date_cannot_be_in_the_past
    if expiration_date.present? && expiration_date < 
Date.today
      errors.add(:expiration_date, "can't be in the 
past")
    end
  end

  def discount_cannot_be_greater_than_total_value
    if discount > total_value
      errors.add(:discount, "can't be greater than total 
value")
    end
  end
end
```

By default, such validations will run every time you call `valid?` or save the object. But it is also possible to control when to run these custom validations by giving an `:on` option to the `validate` method, with either: `:create` or `:update`.

```
class Invoice < ApplicationRecord
  validate :active_customer, on: :create

  def active_customer
    errors.add(:customer_id, "is not active") unless 
customer.active?
  end
end
```

7 Working with Validation Errors

In addition to the `valid?` and `invalid?` methods covered earlier, Rails provides a number of methods for working with the `errors` collection and inquiring about the validity of objects.

The following is a list of the most commonly used methods. Please refer to the `ActiveModel::Errors` documentation for a list of all the available methods.

7.1 `errors`

Returns an instance of the class `ActiveModel::Errors` containing all errors. Each key is the attribute name and the value is an array of strings with all errors.

```
class Person < ApplicationRecord
  validates :name, presence: true, length: { minimum: 
3 }
end

person = Person.new
person.valid? # => false
person.errors.messages
 # => {:name=>["can't be blank", "is too short (minimum 
is 3 characters)"]}

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors.messages # => {}
```

7.2 `errors[]`

`errors[]` is used when you want to check the error messages for a specific attribute. It returns an array of strings with all error messages for the given attribute, each string with one error message. If there are no errors related to the attribute, it returns an empty array.

```
class Person < ApplicationRecord
  validates :name, presence: true, length: { minimum:
3 }
end

person = Person.new(name: "John Doe")
person.valid? # => true
person.errors[:name] # => []

person = Person.new(name: "JD")
person.valid? # => false
person.errors[:name] # => ["is too short (minimum is 3
characters)"]

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3
characters)"]
```

7.3 `errors.add`

The `add` method lets you add an error message related to a particular attribute. It takes as arguments the attribute and the error message.
The `errors.full_messages` method (or its equivalent, `errors.to_a`) returns the error messages in a user-friendly format, with the capitalized attribute name prepended to each message, as shown in the examples below.

```
class Person < ApplicationRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, "cannot contain the characters !
@#%*()_-+=")
  end
end

person = Person.create(name: "!@#")

person.errors[:name]
 # => ["cannot contain the characters !@#%*()_-+="]

person.errors.full_messages
 # => ["Name cannot contain the characters !@#%*()_-+="]
```

An equivalent to `errors#add` is to use << to append a message to the `errors.messages` array for an attribute:

```
class Person < ApplicationRecord
  def a_method_used_for_validation_purposes
    errors.messages[:name] << "cannot contain the 
characters !@#%*()_-+="
  end
end

person = Person.create(name: "!@#")

person.errors[:name]
 # => ["cannot contain the characters !@#%*()_-+="]

person.errors.to_a
 # => ["Name cannot contain the characters !@#%*()_-+="]
```

7.4 `errors.details`

You can specify a validator type to the returned error details hash using the `errors.add` method.

```
class Person < ApplicationRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters)
  end
end

person = Person.create(name: "!@#")

person.errors.details[:name]
# => [{error: :invalid_characters}]
```

To improve the error details to contain the unallowed characters set for instance, you can pass additional keys to `errors.add`.

```
class Person < ApplicationRecord
  def a_method_used_for_validation_purposes
    errors.add(:name, :invalid_characters, not_allowed: 
"!@#%*()_-+=")
  end
end

person = Person.create(name: "!@#")

person.errors.details[:name]
# => [{error: :invalid_characters, not_allowed: "!
@#%*()_-+="}]
```

All built in Rails validators populate the details hash with the corresponding validator type.

7.5 `errors[:base]`

You can add error messages that are related to the object's state as a whole, instead of being related to a specific attribute. You can use this method when you want to say that the object is invalid, no matter the values of its attributes. Since `errors[:base]` is an array, you can simply add a string to it and it will be used as an error message.

```
class Person < ApplicationRecord
  def a_method_used_for_validation_purposes
    errors[:base] << "This person is invalid 
because ..."
  end
end
```

7.6 `errors.clear`

The `clear` method is used when you intentionally want to clear all the messages in the `errors` collection. Of course, calling `errors.clear` upon an invalid object won't actually make it valid: the `errors` collection will now be empty, but the next time you call `valid?` or any method that tries to save this object to the database, the validations will run again. If any of the validations fail, the `errors` collection will be filled again.

```
class Person < ApplicationRecord
  validates :name, presence: true, length: { minimum: 
3 }
end

person = Person.new
person.valid? # => false
person.errors[:name]
 # => ["can't be blank", "is too short (minimum is 3 
characters)"]

person.errors.clear
person.errors.empty? # => true

person.save # => false

person.errors[:name]
# => ["can't be blank", "is too short (minimum is 3 
characters)"]
```

7.7 `errors.size`

The `size` method returns the total number of error messages for the object.

```
class Person < ApplicationRecord
  validates :name, presence: true, length: { minimum: 3 }
end

person = Person.new
person.valid? # => false
person.errors.size # => 2

person = Person.new(name: "Andrea", email: "andrea@example.com")
person.valid? # => true
person.errors.size # => 0
```

8 Displaying Validation Errors in Views

Once you've created a model and added validations, if that model is created via a web form, you probably want to display an error message when one of the validations fail.

Because every application handles this kind of thing differently, Rails does not include any view helpers to help you generate these messages directly. However, due to the rich number of methods Rails gives you to interact with validations in general, it's fairly easy to build your own. In addition, when generating a scaffold, Rails will put some ERB into the `_form.html.erb` that it generates that displays the full list of errors on that model.

Assuming we have a model that's been saved in an instance variable named `@article`, it looks like this:

```
<% if @article.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(@article.errors.count, "error") %> prohibited this article from being saved:</h2>

    <ul>
    <% @article.errors.full_messages.each do |msg| %>
      <li><%= msg %></li>
    <% end %>
    </ul>
  </div>
<% end %>
```

Furthermore, if you use the Rails form helpers to generate your forms, when a validation error occurs on a field, it will generate an extra `<div>` around the entry.

```
<div class="field_with_errors">
 <input id="article_title" name="article[title]" 
size="30" type="text" value="">
</div>
```
You can then style this div however you'd like. The default scaffold that Rails generates, for example, adds this CSS rule:
```
.field_with_errors {
  padding: 2px;
  background-color: red;
  display: table;
}
```
This means that any field with an error ends up with a 2 pixel red border.

1 Migration Overview

Migrations are a convenient way to alter your database schema over time in a consistent and easy way. They use a Ruby DSL so that you don't have to write SQL by hand, allowing your schema and changes to be database independent.
You can think of each migration as being a new 'version' of the database. A schema starts off with nothing in it, and each migration modifies it to add or remove tables, columns, or entries. Active Record knows how to update your schema along this timeline, bringing it from whatever point it is in the history to the latest version. Active Record will also update your `db/schema.rb` file to match the up-to-date structure of your database.
Here's an example of a migration:

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

This migration adds a table called `products` with a string column called `name` and a text column called `description`. A primary key column called `id` will also be added implicitly, as it's the default primary key for all Active Record models. The `timestamps` macro adds two columns, `created_at` and `updated_at`. These special columns are automatically managed by Active Record if they exist.
Note that we define the change that we want to happen moving forward in time. Before this migration is run, there will be no table. After, the table will exist. Active Record knows how to reverse this migration as well: if we roll this migration back, it will remove the table.
On databases that support transactions with statements that change the schema, migrations are wrapped in a transaction. If the database does not

support this then when a migration fails the parts of it that succeeded will not be rolled back. You will have to rollback the changes that were made by hand. There are certain queries that can't run inside a transaction. If your adapter supports DDL transactions you can use `disable_ddl_transaction!` to disable them for a single migration.

If you wish for a migration to do something that Active Record doesn't know how to reverse, you can use `reversible`:

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  def change
    reversible do |dir|
      change_table :products do |t|
        dir.up   { t.change :price, :string }
        dir.down { t.change :price, :integer }
      end
    end
  end
end
```

Alternatively, you can use `up` and `down` instead of `change`:

```
class ChangeProductsPrice < ActiveRecord::Migration[5.0]
  def up
    change_table :products do |t|
      t.change :price, :string
    end
  end

  def down
    change_table :products do |t|
      t.change :price, :integer
    end
  end
end
```

2 Creating a Migration

2.1 Creating a Standalone Migration

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_products.rb`, that is to say a UTC timestamp identifying the migration followed by an underscore followed by the name of the migration.

The name of the migration class (CamelCased version) should match the latter part of the file name. For example `20080906120000_create_products.rb` should define class `CreateProducts` and `20080906120001_add_details_to_products.rb` should define `AddDetailsToProducts`.

Rails uses this timestamp to determine which migration should be run and in what order, so if you're copying a migration from another application or generate a file yourself, be aware of its position in the order.

Of course, calculating timestamps is no fun, so Active Record provides a generator to handle making it for you:

```
$ bin/rails generate migration AddPartNumberToProducts
```

This will create an empty but appropriately named migration:

```
class AddPartNumberToProducts <
ActiveRecord::Migration[5.0]
  def change
  end
end
```

If the migration name is of the form "AddXXXToYYY" or "RemoveXXXFromYYY" and is followed by a list of column names and types then a migration containing the appropriate `add_column` and `remove_column` statements will be created.

```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string
```

will generate

```
class AddPartNumberToProducts <
ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
  end
end
```

If you'd like to add an index on the new column, you can do that as well:

```
$ bin/rails generate migration AddPartNumberToProducts
part_number:string:index
```

will generate

```
class AddPartNumberToProducts <
ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_index :products, :part_number
  end
end
```

Similarly, you can generate a migration to remove a column from the command line:

```
$ bin/rails generate migration
RemovePartNumberFromProducts part_number:string
```

generates

```
class RemovePartNumberFromProducts <
ActiveRecord::Migration[5.0]
  def change
    remove_column :products, :part_number, :string
  end
end
```

You are not limited to one magically generated column. For example:

```
$ bin/rails generate migration AddDetailsToProducts
part_number:string price:decimal
```

generates

```
class AddDetailsToProducts <
ActiveRecord::Migration[5.0]
  def change
    add_column :products, :part_number, :string
    add_column :products, :price, :decimal
  end
end
```

If the migration name is of the form "CreateXXX" and is followed by a list of column names and types then a migration creating the table XXX with the columns listed will be generated.

For example:

```
$ bin/rails generate migration CreateProducts
name:string part_number:string
```

generates

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.string :part_number
    end
  end
end
```

As always, what has been generated for you is just a starting point. You can add or remove from it as you see fit by editing the `db/migrate/YYYYMMDDHHMMSS_add_details_to_products.rb` file.

Also, the generator accepts column type as `references` (also available as `belongs_to`). For instance:

```
$ bin/rails generate migration AddUserRefToProducts
user:references
```

generates

```
class AddUserRefToProducts <
ActiveRecord::Migration[5.0]
  def change
    add_reference :products, :user, foreign_key: true
  end
end
```

This migration will create a `user_id` column and appropriate index. For more `add_reference` options, visit the API documentation.

There is also a generator which will produce join tables if `JoinTable` is part of the name:

```
$ bin/rails g migration CreateJoinTableCustomerProduct
customer product
```

will produce the following migration:

```
class CreateJoinTableCustomerProduct <
ActiveRecord::Migration[5.0]
  def change
    create_join_table :customers, :products do |t|
      # t.index [:customer_id, :product_id]
      # t.index [:product_id, :customer_id]
    end
  end
end
```

2.2 Model Generators

The model and scaffold generators will create migrations appropriate for adding a new model. This migration will already contain instructions for creating the relevant table. If you tell Rails what columns you want, then statements for adding these columns will also be created. For example, running:

```
$ bin/rails generate model Product name:string
description:text
```

will create a migration that looks like this

```
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    create_table :products do |t|
      t.string :name
      t.text :description

      t.timestamps
    end
  end
end
```

You can append as many column name/type pairs as you want.

2.3 Passing Modifiers

Some commonly used type modifiers can be passed directly on the command line. They are enclosed by curly braces and follow the field type:

For instance, running:

```
$ bin/rails generate migration AddDetailsToProducts
'price:decimal{5,2}' supplier:references{polymorphic}
```

will produce a migration that looks like this

```
class AddDetailsToProducts <
ActiveRecord::Migration[5.0]
  def change
    add_column :products, :price, :decimal, precision:
5, scale: 2
    add_reference :products, :supplier, polymorphic:
true
  end
end
```

Have a look at the generators help output for further details.

3 Writing a Migration

Once you have created your migration using one of the generators it's time to get to work!

3.1 Creating a Table

The `create_table` method is one of the most fundamental, but most of the time, will be generated for you from using a model or scaffold generator. A typical use would be

```
create_table :products do |t|
  t.string :name
end
```

which creates a `products` table with a column called `name` (and as discussed below, an implicit `id` column).

By default, `create_table` will create a primary key called `id`. You can change the name of the primary key with the `:primary_key` option (don't forget to update the corresponding model) or, if you don't want a primary key at all, you can pass the option `id: false`. If you need to pass database specific options you can place an SQL fragment in the `:options` option. For example:

```
create_table :products, options: "ENGINE=BLACKHOLE" do |
t|
  t.string :name, null: false
end
```

will append `ENGINE=BLACKHOLE` to the SQL statement used to create the table.

Also you can pass the `:comment` option with any description for the table that will be stored in database itself and can be viewed with database administration tools, such as MySQL Workbench or PgAdmin III. It's highly recommended to specify comments in migrations for applications with large databases as it helps people to understand data model and generate documentation. Currently only the MySQL and PostgreSQL adapters support comments.

3.2 Creating a Join Table

The migration method `create_join_table` creates an HABTM (has and belongs to many) join table. A typical use would be:

```
create_join_table :products, :categories
```

which creates a `categories_products` table with two columns called `category_id` and `product_id`. These columns have the option `:null` set to `false` by default. This can be overridden by specifying the `:column_options` option:

```
create_join_table :products, :categories,
column_options: { null: true }
```

By default, the name of the join table comes from the union of the first two arguments provided to create_join_table, in alphabetical order. To customize the name of the table, provide a `:table_name` option:

```
create_join_table :products, :categories,
table_name: :categorization
```

creates a `categorization` table.

`create_join_table` also accepts a block, which you can use to add indices (which are not created by default) or additional columns:

```
create_join_table :products, :categories do |t|
  t.index :product_id
  t.index :category_id
end
```

3.3 Changing Tables

A close cousin of `create_table` is `change_table`, used for changing existing tables. It is used in a similar fashion to `create_table` but the object yielded to the block knows more tricks. For example:

```
change_table :products do |t|
  t.remove :description, :name
  t.string :part_number
  t.index :part_number
  t.rename :upccode, :upc_code
end
```

removes the `description` and `name` columns, creates a `part_number` string column and adds an index on it. Finally it renames the `upccode` column.

3.4 Changing Columns

Like the `remove_column` and `add_column` Rails provides the `change_column` migration method.

```
change_column :products, :part_number, :text
```

This changes the column `part_number` on products table to be a `:text` field. Note that `change_column` command is irreversible.

Besides `change_column`, the `change_column_null` and `change_column_default` methods are used specifically to change a not null constraint and default values of a column.

```
change_column_null :products, :name, false
change_column_default :products, :approved, from: true, 
to: false
```

This sets `:name` field on products to a `NOT NULL` column and the default value of the `:approved` field from true to false.

Note: You could also write the above `change_column_default` migration as `change_column_default :products, :approved, false`, but unlike the previous example, this would make your migration irreversible.

3.5 Column Modifiers

Column modifiers can be applied when creating or changing a column:

- `limit` Sets the maximum size of the `string/text/binary/integer` fields.
- `precision` Defines the precision for the `decimal` fields, representing the total number of digits in the number.

- `scale` Defines the scale for the `decimal` fields, representing the number of digits after the decimal point.
- `polymorphic` Adds a `type` column for `belongs_to` associations.
- `null` Allows or disallows `NULL` values in the column.
- `default` Allows to set a default value on the column. Note that if you are using a dynamic value (such as a date), the default will only be calculated the first time (i.e. on the date the migration is applied).
- `index` Adds an index for the column.
- `comment` Adds a comment for the column.

Some adapters may support additional options; see the adapter specific API docs for further information.

`null` and `default` cannot be specified via command line.

3.6 Foreign Keys

While it's not required you might want to add foreign key constraints to guarantee referential integrity.

```
add_foreign_key :articles, :authors
```

This adds a new foreign key to the `author_id` column of the `articles` table. The key references the `id` column of the `authors` table. If the column names can not be derived from the table names, you can use the `:column` and `:primary_key` options.

Rails will generate a name for every foreign key starting with `fk_rails_` followed by 10 characters which are deterministically generated from the `from_table` and `column`. There is a `:name` option to specify a different name if needed.

Active Record only supports single column foreign keys. `execute` and `structure.sql` are required to use composite foreign keys. See Schema Dumping and You.

Removing a foreign key is easy as well:

```
# let Active Record figure out the column name
remove_foreign_key :accounts, :branches

# remove foreign key for a specific column
remove_foreign_key :accounts, column: :owner_id

# remove foreign key by name
remove_foreign_key :accounts, name: :special_fk_name
```

3.7 When Helpers aren't Enough

If the helpers provided by Active Record aren't enough you can use the `execute` method to execute arbitrary SQL:

```
Product.connection.execute("UPDATE products SET price = 'free' WHERE 1=1")
```

For more details and examples of individual methods, check the API documentation. In particular the documentation for ActiveRecord::ConnectionAdapters::SchemaStatements (which provides the methods available in the `change`, `up` and `down` methods), ActiveRecord::ConnectionAdapters::TableDefinition (which provides the methods available on the object yielded by `create_table`) and ActiveRecord::ConnectionAdapters::Table (which provides the methods available on the object yielded by `change_table`).

3.8 Using the change Method

The `change` method is the primary way of writing migrations. It works for the majority of cases, where Active Record knows how to reverse the migration automatically. Currently, the `change` method supports only these migration definitions:

- add_column
- add_foreign_key
- add_index
- add_reference
- add_timestamps
- change_column_default (must supply a :from and :to option)
- change_column_null
- create_join_table
- create_table
- disable_extension
- drop_join_table
- drop_table (must supply a block)
- enable_extension
- remove_column (must supply a type)
- remove_foreign_key (must supply a second table)
- remove_index
- remove_reference
- remove_timestamps
- rename_column
- rename_index
- rename_table

`change_table` is also reversible, as long as the block does not call `change`, `change_default` or `remove`.

`remove_column` is reversible if you supply the column type as the third argument. Provide the original column options too, otherwise Rails can't recreate the column exactly when rolling back:

```
remove_column :posts, :slug, :string, null: false,
default: '', index: true
```

If you're going to need to use any other methods, you should use `reversible` or write the `up` and `down` methods instead of using the `change` method.

3.9 Using `reversible`

Complex migrations may require processing that Active Record doesn't know how to reverse. You can use `reversible` to specify what to do when running a migration and what else to do when reverting it. For example:

```
class ExampleMigration < ActiveRecord::Migration[5.0]
  def change
    create_table :distributors do |t|
      t.string :zipcode
    end

    reversible do |dir|
      dir.up do
        # add a CHECK constraint
        execute <<-SQL
          ALTER TABLE distributors
            ADD CONSTRAINT zipchk
              CHECK (char_length(zipcode) = 5) NO
INHERIT;
        SQL
      end
      dir.down do
        execute <<-SQL
          ALTER TABLE distributors
            DROP CONSTRAINT zipchk
        SQL
      end
    end

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end
end
```

Using `reversible` will ensure that the instructions are executed in the right order too. If the previous example migration is reverted, the `down` block will be run after the `home_page_url` column is removed and right before the table `distributors` is dropped.

Sometimes your migration will do something which is just plain irreversible; for example, it might destroy some data. In such cases, you can raise `ActiveRecord::IrreversibleMigration` in your `down` block. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

3.10 Using the up/down Methods

You can also use the old style of migration using `up` and `down` methods instead of the `change` method. The `up` method should describe the transformation you'd like to make to your schema, and the `down` method of your migration should revert the transformations done by the `up` method. In other words, the database schema should be unchanged if you do an `up` followed by a `down`. For example, if you create a table in the `up` method, you should drop it in the `down` method. It is wise to perform the transformations in precisely the reverse order they were made in the `up` method. The example in the `reversible` section is equivalent to:

```
class ExampleMigration < ActiveRecord::Migration[5.0]
  def up
    create_table :distributors do |t|
      t.string :zipcode
    end

    # add a CHECK constraint
    execute <<-SQL
      ALTER TABLE distributors
        ADD CONSTRAINT zipchk
        CHECK (char_length(zipcode) = 5);
    SQL

    add_column :users, :home_page_url, :string
    rename_column :users, :email, :email_address
  end

  def down
    rename_column :users, :email_address, :email
    remove_column :users, :home_page_url

    execute <<-SQL
      ALTER TABLE distributors
        DROP CONSTRAINT zipchk
    SQL

    drop_table :distributors
  end
end
```

If your migration is irreversible, you should raise `ActiveRecord::IrreversibleMigration` from your `down` method. If someone tries to revert your migration, an error message will be displayed saying that it can't be done.

3.11 Reverting Previous Migrations

You can use Active Record's ability to rollback migrations using the `revert` method:

```
require_relative '20121212123456_example_migration'

class FixupExampleMigration < 
ActiveRecord::Migration[5.0]
  def change
    revert ExampleMigration

    create_table(:apples) do |t|
      t.string :variety
    end
  end
end
```

The `revert` method also accepts a block of instructions to reverse. This could be useful to revert selected parts of previous migrations. For example, let's imagine that `ExampleMigration` is committed and it is later decided it would be best to use Active Record validations, in place of the `CHECK` constraint, to verify the zipcode.

```
class DontUseConstraintForZipcodeValidationMigration <
ActiveRecord::Migration[5.0]
  def change
    revert do
      # copy-pasted code from ExampleMigration
      reversible do |dir|
        dir.up do
          # add a CHECK constraint
          execute <<-SQL
            ALTER TABLE distributors
              ADD CONSTRAINT zipchk
                CHECK (char_length(zipcode) = 5);
          SQL
        end
        dir.down do
          execute <<-SQL
            ALTER TABLE distributors
              DROP CONSTRAINT zipchk
          SQL
        end
      end

      # The rest of the migration was ok
    end
  end
end
```

The same migration could also have been written without using `revert` but this would have involved a few more steps: reversing the order of `create_table` and `reversible`, replacing `create_table` by `drop_table`, and finally replacing `up` by `down` and vice-versa. This is all taken care of by `revert`.

If you want to add check constraints like in the examples above, you will have to use `structure.sql` as dump method. See Schema Dumping and You.

4 Running Migrations

Rails provides a set of bin/rails tasks to run certain sets of migrations.

The very first migration related bin/rails task you will use will probably be `rails db:migrate`. In its most basic form it just runs the `change` or `up` method for all the migrations that have not yet been run. If there are no such migrations, it exits. It will run these migrations in order based on the date of the migration.

Note that running the `db:migrate` task also invokes the `db:schema:dump` task, which will update your `db/schema.rb` file to match the structure of your database.
If you specify a target version, Active Record will run the required migrations (change, up, down) until it has reached the specified version. The version is the numerical prefix on the migration's filename. For example, to migrate to version 20080906120000 run:

```
$ bin/rails db:migrate VERSION=20080906120000
```

If version 20080906120000 is greater than the current version (i.e., it is migrating upwards), this will run the `change` (or `up`) method on all migrations up to and including 20080906120000, and will not execute any later migrations. If migrating downwards, this will run the `down` method on all the migrations down to, but not including, 20080906120000.

4.1 Rolling Back

A common task is to rollback the last migration. For example, if you made a mistake in it and wish to correct it. Rather than tracking down the version number associated with the previous migration you can run:

```
$ bin/rails db:rollback
```

This will rollback the latest migration, either by reverting the `change` method or by running the `down` method. If you need to undo several migrations you can provide a `STEP` parameter:

```
$ bin/rails db:rollback STEP=3
```

will revert the last 3 migrations.
The `db:migrate:redo` task is a shortcut for doing a rollback and then migrating back up again. As with the `db:rollback` task, you can use the `STEP` parameter if you need to go more than one version back, for example:

```
$ bin/rails db:migrate:redo STEP=3
```

Neither of these bin/rails tasks do anything you could not do with `db:migrate`. They are simply more convenient, since you do not need to explicitly specify the version to migrate to.

4.2 Setup the Database

The `rails db:setup` task will create the database, load the schema and initialize it with the seed data.

4.3 Resetting the Database

The `rails db:reset` task will drop the database and set it up again. This is functionally equivalent to `rails db:drop db:setup`.
This is not the same as running all the migrations. It will only use the contents of the current `db/schema.rb` or `db/structure.sql` file. If a migration can't be rolled back, `rails db:reset` may not help you. To find out more about dumping the schema see Schema Dumping and You section.

4.4 Running Specific Migrations

If you need to run a specific migration up or down, the `db:migrate:up` and `db:migrate:down` tasks will do that. Just specify the appropriate version and the corresponding migration will have its `change`, `up` or `down` method invoked, for example:

```
$ bin/rails db:migrate:up VERSION=20080906120000
```

will run the 20080906120000 migration by running the `change` method (or the `up` method). This task will first check whether the migration is already performed and will do nothing if Active Record believes that it has already been run.

4.5 Running Migrations in Different Environments

By default running `bin/rails db:migrate` will run in the `development` environment. To run migrations against another environment you can specify it using the `RAILS_ENV` environment variable while running the command. For example to run migrations against the `test` environment you could run:

```
$ bin/rails db:migrate RAILS_ENV=test
```

4.6 Changing the Output of Running Migrations

By default migrations tell you exactly what they're doing and how long it took. A migration creating a table and adding an index might produce output like this

```
==  CreateProducts: migrating
=================================================
-- create_table(:products)
   -> 0.0028s
==  CreateProducts: migrated (0.0028s)
========================================
```

Several methods are provided in migrations that allow you to control all this:

Method	Purpose
suppress_messages	Takes a block as an argument and suppresses any output generated by the block.
say	Takes a message argument and outputs it as is. A second boolean argument can be passed to specify whether to indent or not.
say_with_time	Outputs text along with how long it took to run its block. If the block returns an integer it assumes it is the number of rows affected.

For example, this migration:

```ruby
class CreateProducts < ActiveRecord::Migration[5.0]
  def change
    suppress_messages do
      create_table :products do |t|
        t.string :name
        t.text :description
        t.timestamps
      end
    end

    say "Created a table"

    suppress_messages {add_index :products, :name}
    say "and an index!", true

    say_with_time 'Waiting for a while' do
      sleep 10
      250
    end
  end
end
```

generates the following output

```
==  CreateProducts: migrating 
=================================================
-- Created a table
   -> and an index!
-- Waiting for a while
   -> 10.0013s
   -> 250 rows
==  CreateProducts: migrated (10.0054s) 
=======================================
```

If you want Active Record to not output anything, then running `rails db:migrate`
`VERBOSE=false` will suppress all output.

5 Changing Existing Migrations

Occasionally you will make a mistake when writing a migration. If you have already run the migration, then you cannot just edit the migration and run the migration again: Rails thinks it has already run the migration and so will do nothing when you run `rails db:migrate`. You must rollback the migration (for example with `bin/rails db:rollback`), edit your migration and then run `rails db:migrate` to run the corrected version.

In general, editing existing migrations is not a good idea. You will be creating extra work for yourself and your co-workers and cause major headaches if the existing version of the migration has already been run on production machines. Instead, you should write a new migration that performs the changes you require. Editing a freshly generated migration that has not yet been committed to source control (or, more generally, which has not been propagated beyond your development machine) is relatively harmless.

The `revert` method can be helpful when writing a new migration to undo previous migrations in whole or in part (see Reverting Previous Migrations above).

6 Schema Dumping and You

6.1 What are Schema Files for?

Migrations, mighty as they may be, are not the authoritative source for your database schema. That role falls to either `db/schema.rb` or an SQL file which Active Record generates by examining the database. They are not designed to be edited, they just represent the current state of the database.

There is no need (and it is error prone) to deploy a new instance of an app by replaying the entire migration history. It is much simpler and faster to just load into the database a description of the current schema.

For example, this is how the test database is created: the current development database is dumped (either to `db/schema.rb` or `db/structure.sql`) and then loaded into the test database.

Schema files are also useful if you want a quick look at what attributes an Active Record object has. This information is not in the model's code and is frequently spread across several migrations, but the information is nicely summed up in the schema file. The annotate_models gem automatically adds and updates comments at the top of each model summarizing the schema if you desire that functionality.

6.2 Types of Schema Dumps

There are two ways to dump the schema. This is set in `config/application.rb` by the `config.active_record.schema_format` setting, which may be either `:sql` or `:ruby`.

If `:ruby` is selected, then the schema is stored in `db/schema.rb`. If you look at this file you'll find that it looks an awful lot like one very big migration:

```
ActiveRecord::Schema.define(version: 20080906171750) do
  create_table "authors", force: true do |t|
    t.string   "name"
    t.datetime "created_at"
    t.datetime "updated_at"
  end

  create_table "products", force: true do |t|
    t.string   "name"
    t.text     "description"
    t.datetime "created_at"
    t.datetime "updated_at"
    t.string   "part_number"
  end
end
```

In many ways this is exactly what it is. This file is created by inspecting the database and expressing its structure using `create_table`, `add_index`, and so on. Because this is database-independent, it could be loaded into any database that Active Record supports. This could be very useful if you were to distribute an application that is able to run against multiple databases.

`db/schema.rb` cannot express database specific items such as triggers, sequences, stored procedures or check constraints, etc. Please note that while custom SQL statements can be run in migrations, these statements cannot be reconstituted by the schema dumper. If you are using features like this, then you should set the schema format to `:sql`.

Instead of using Active Record's schema dumper, the database's structure will be dumped using a tool specific to the database (via the `db:structure:dump` rails task) into `db/structure.sql`. For example, for PostgreSQL, the `pg_dump` utility is used. For MySQL and MariaDB, this file will contain the output of `SHOW CREATE TABLE` for the various tables.

Loading these schemas is simply a question of executing the SQL statements they contain. By definition, this will create a perfect copy of the database's structure. Using the `:sql` schema format will, however, prevent loading the schema into a RDBMS other than the one used to create it.

6.3 Schema Dumps and Source Control

Because schema dumps are the authoritative source for your database schema, it is strongly recommended that you check them into source control.

`db/schema.rb` contains the current version number of the database. This ensures conflicts are going to happen in the case of a merge where both branches touched the schema. When that happens, solve conflicts manually, keeping the highest version number of the two.

7 Active Record and Referential Integrity

The Active Record way claims that intelligence belongs in your models, not in the database. As such, features such as triggers or constraints, which push some of that intelligence back into the database, are not heavily used.
Validations such as `validates :foreign_key, uniqueness: true` are one way in which models can enforce data integrity. The `:dependent` option on associations allows models to automatically destroy child objects when the parent is destroyed. Like anything which operates at the application level, these cannot guarantee referential integrity and so some people augment them with foreign key constraints in the database.
Although Active Record does not provide all the tools for working directly with such features, the `execute` method can be used to execute arbitrary SQL.

8 Migrations and Seed Data

The main purpose of Rails' migration feature is to issue commands that modify the schema using a consistent process. Migrations can also be used to add or modify data. This is useful in an existing database that can't be destroyed and recreated, such as a production database.

```
class AddInitialProducts < ActiveRecord::Migration[5.0]
  def up
    5.times do |i|
      Product.create(name: "Product ##{i}", description:
"A product.")
    end
  end

  def down
    Product.delete_all
  end
end
```

To add initial data after a database is created, Rails has a built-in 'seeds' feature that makes the process quick and easy. This is especially useful when reloading the database frequently in development and test environments. It's easy to get started with this feature: just fill up `db/seeds.rb` with some Ruby code, and run `rails db:seed`:

```
5.times do |i|
  Product.create(name: "Product ##{i}", description: "A
product.")
end
```

This is generally a much cleaner way to set up the database of a blank application.

1 What Does a Controller Do?

A controller is a middleman between models and views.

It makes the model data available to the view so it can display that data to the user, and it saves or updates user data to the model.

2 Controller Naming Convention

The naming convention of controllers in Rails **favors pluralization** of the last word in the controller's name, although it is not strictly required (e.g. `ApplicationController`).

For example, `ClientsController` is preferable to `ClientController`,

`SiteAdminsController` is preferable to `SiteAdminController` or `SitesAdminsController`, and so on.

Following this convention will allow you to use the default route generators (e.g. `resources`, etc) without needing to qualify each `:path` or `:controller`, and will keep URL and path helpers' usage consistent throughout your application.

The controller naming convention differs from the naming convention of models, which are expected to be named in singular form.

3 Methods and Actions

A controller is a Ruby class which inherits from `ApplicationController` and has methods just like any other class. When your application receives a request, the routing will determine which controller and action to run, then Rails creates an instance of that controller and runs the method with the same name as the action.

```
class ClientsController < ApplicationController
  def new
  end
end
```

As an example, if a user goes to `/clients/new` in your application to add a new client, Rails will create an instance of `ClientsController` and call its `new` method. Note that the empty method from the example above would work just fine because Rails will by default render the `new.html.erb` view unless the action says otherwise. The `new` method could make available to the view a `@client` instance variable by creating a new `Client`:

```
def new
  @client = Client.new
end
```

The Layouts & Rendering Guide explains this in more detail.

`ApplicationController` inherits from `ActionController::Base`, which defines a number of helpful methods. This guide will cover some of these, but if you're curious to see what's in there, you can see all of them in the API documentation or in the source itself.

Only public methods are callable as actions. It is a best practice to lower the visibility of methods (with `private` or `protected`) which are not intended to be actions, like auxiliary methods or filters.

4 Parameters

You will probably want to access data sent in by the user or other parameters in your controller actions. There are two kinds of parameters possible in a web application. The first are parameters that are sent as part of the URL, called query string parameters. The query string is everything after "?" in the URL. The second type of parameter is usually referred to as POST data. This information usually comes from an HTML form which has been filled in by the user. It's called POST data because it can only be sent as part of an HTTP POST request. Rails does not make any distinction between query string parameters and POST parameters, and both are available in the `params` hash in your controller:

```
class ClientsController < ApplicationController
  # This action uses query string parameters because it 
gets run
  # by an HTTP GET request, but this does not make any 
difference
  # to the way in which the parameters are accessed. The 
URL for
  # this action would look like this in order to list 
activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.inactivated
    end
  end

  # This action uses POST parameters. They are most 
likely coming
  # from an HTML form which the user has submitted. The 
URL for
  # this RESTful request will be "/clients", and the 
data will be
  # sent as part of the request body.
  def create
    @client = Client.new(params[:client])
    if @client.save
      redirect_to @client
    else
      # This line overrides the default rendering 
behavior, which
      # would have been to render the "create" view.
      render "new"
    end
  end
end
```

4.1 Hash and Array Parameters

The `params` hash is not limited to one-dimensional keys and values. It can contain nested arrays and hashes. To send an array of values, append an empty pair of square brackets "[]" to the key name:

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

The actual URL in this example will be encoded as "/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3" as the "[" and "]" characters are not allowed in URLs. Most of the time you don't have to worry about this because the browser will encode it for you, and Rails will decode it automatically, but if you ever find yourself having to send those requests to the server manually you should keep this in mind.

The value of `params[:ids]` will now be `["1", "2", "3"]`. Note that parameter values are always strings; Rails makes no attempt to guess or cast the type.

Values such as `[nil]` or `[nil, nil, ...]` in `params` are replaced with `[]` for security reasons by default. See Security Guide for more information.

To send a hash, you include the key name inside the brackets:

```
<form accept-charset="UTF-8" action="/clients"
method="post">
  <input type="text" name="client[name]" value="Acme" />
  <input type="text" name="client[phone]"
value="12345" />
  <input type="text" name="client[address][postcode]"
value="12345" />
  <input type="text" name="client[address][city]"
value="Carrot City" />
</form>
```

When this form is submitted, the value of `params[:client]` will be `{ "name" => "Acme", "phone" => "12345", "address" => { "postcode" => "12345", "city" => "Carrot City" } }`. Note the nested hash in `params[:client][:address]`.

The `params` object acts like a Hash, but lets you use symbols and strings interchangeably as keys.

4.2 JSON parameters

If you're writing a web service application, you might find yourself more comfortable accepting parameters in JSON format. If the "Content-Type" header of your request is set to "application/json", Rails will automatically load your parameters into the `params` hash, which you can access as you would normally.

So for example, if you are sending this JSON content:

```
{ "company": { "name": "acme", "address": "123 Carrot
Street" } }
```

Your controller will receive `params[:company]` as `{ "name" => "acme", "address" => "123 Carrot Street" }`.

Also, if you've turned on `config.wrap_parameters` in your initializer or called `wrap_parameters` in your controller, you can safely omit the root element in the JSON parameter. In this case, the parameters will be cloned and wrapped with a key chosen based on your controller's name. So the above JSON request can be written as:

```
{ "name": "acme", "address": "123 Carrot Street" }
```

And, assuming that you're sending the data to `CompaniesController`, it would then be wrapped within the `:company` key like this:

```
{ name: "acme", address: "123 Carrot Street", company: {
name: "acme", address: "123 Carrot Street" } }
```

You can customize the name of the key or specific parameters you want to wrap by consulting the API documentation

Support for parsing XML parameters has been extracted into a gem named `actionpack-xml_parser`.

4.3 Routing Parameters

The `params` hash will always contain the `:controller` and `:action` keys, but you should use the methods `controller_name` and `action_name` instead to access these values. Any other parameters defined by the routing, such as `:id`, will also be available. As an example, consider a listing of clients where the list can show either active or inactive clients. We can add a route which captures the `:status` parameter in a "pretty" URL:

```
get '/clients/:status' => 'clients#index', foo: 'bar'
```

In this case, when a user opens the URL `/clients/active`, `params[:status]` will be set to "active". When this route is used, `params[:foo]` will also be set to "bar", as if it were passed in the query string. Your controller will also receive `params[:action]` as "index" and `params[:controller]` as "clients".

4.4 `default_url_options`

You can set global default parameters for URL generation by defining a method called `default_url_options` in your controller. Such a method must return a hash with the desired defaults, whose keys must be symbols:

```
class ApplicationController < ActionController::Base
  def default_url_options
    { locale: I18n.locale }
  end
end
```

These options will be used as a starting point when generating URLs, so it's possible they'll be overridden by the options passed to `url_for` calls.

If you define `default_url_options` in `ApplicationController`, as in the example above, these defaults will be used for all URL generation. The method can also be defined in a specific controller, in which case it only affects URLs generated there.

In a given request, the method is not actually called for every single generated URL; for performance reasons, the returned hash is cached, there is at most one invocation per request.

4.5 Strong Parameters

With strong parameters, Action Controller parameters are forbidden to be used in Active Model mass assignments until they have been whitelisted. This means that you'll have to make a conscious decision about which attributes to allow for mass update. This is a better security practice to help prevent accidentally allowing users to update sensitive model attributes.

In addition, parameters can be marked as required and will flow through a predefined raise/rescue flow that will result in a 400 Bad Request being returned if not all required parameters are passed in.

```
class PeopleController < ActionController::Base
  # This will raise an 
ActiveModel::ForbiddenAttributesError exception
  # because it's using mass assignment without an 
explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's 
a person key
  # in the parameters, otherwise it'll raise a
  # ActionController::ParameterMissing exception, which 
will get
  # caught by ActionController::Base and turned into a 
400 Bad
  # Request error.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the 
permissible parameters
    # is just a good pattern since you'll be able to 
reuse the same
    # permit list between create and update. Also, you 
can specialize
    # this method with per-user checking of permissible 
attributes.
    def person_params
      params.require(:person).permit(:name, :age)
    end
end
```

4.5.1 Permitted Scalar Values

Given

```
params.permit(:id)
```

the key `:id` will pass the whitelisting if it appears in `params` and it has a permitted scalar value associated. Otherwise, the key is going to be filtered out, so arrays, hashes, or any other objects cannot be injected.

The permitted scalar types are `String`, `Symbol`, `NilClass`, `Numeric`, `TrueClass`, `FalseClass`, `Date`, `Time`, `DateTime`, `StringIO`, `IO`, `ActionDispatch::Http::UploadedFile`, and `Rack::Test::UploadedFile`.
To declare that the value in `params` must be an array of permitted scalar values, map the key to an empty array:

```
params.permit(id: [])
```

Sometimes it is not possible or convenient to declare the valid keys of a hash parameter or its internal structure. Just map to an empty hash:

```
params.permit(preferences: {})
```

but be careful because this opens the door to arbitrary input. In this case, `permit` ensures values in the returned structure are permitted scalars and filters out anything else.
To whitelist an entire hash of parameters, the `permit!` method can be used:

```
params.require(:log_entry).permit!
```

This marks the `:log_entry` parameters hash and any sub-hash of it as permitted and does not check for permitted scalars, anything is accepted. Extreme care should be taken when using `permit!`, as it will allow all current and future model attributes to be mass-assigned.

4.5.2 Nested Parameters

You can also use `permit` on nested parameters, like:

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ], hobbies: 
[] }])
```

This declaration whitelists the `name`, `emails`, and `friends` attributes. It is expected that `emails` will be an array of permitted scalar values, and that `friends` will be an array of resources with specific attributes: they should have a `name` attribute (any permitted scalar values allowed), a `hobbies` attribute as an array of permitted scalar values, and a `family` attribute which is restricted to having a `name` (any permitted scalar values allowed here, too).

4.5.3 More Examples

You may want to also use the permitted attributes in your `new` action. This raises the problem that you can't use `require` on the root key because, normally, it does not exist when calling `new`:

```
# using `fetch` you can supply a default and use
# the Strong Parameters API from there.
params.fetch(:blog, {}).permit(:title, :author)
```

The model class method `accepts_nested_attributes_for` allows you to update and destroy associated records. This is based on the `id` and `_destroy` parameters:

```
# permit :id and :_destroy
params.require(:author).permit(:name, books_attributes: 
[:title, :id, :_destroy])
```

Hashes with integer keys are treated differently, and you can declare the attributes as if they were direct children. You get these kinds of parameters when you use `accepts_nested_attributes_for` in combination with a `has_many` association:

```
# To whitelist the following data:
# {"book" => {"title" => "Some Book",
#             "chapters_attributes" => { "1" => {"title" 
=> "First Chapter"},
#                                        "2" => {"title" 
=> "Second Chapter"}}}}

params.require(:book).permit(:title, 
chapters_attributes: [:title])
```

4.5.4 Outside the Scope of Strong Parameters

The strong parameter API was designed with the most common use cases in mind. It is not meant as a silver bullet to handle all of your whitelisting problems. However, you can easily mix the API with your own code to adapt to your situation.

Imagine a scenario where you have parameters representing a product name and a hash of arbitrary data associated with that product, and you want to whitelist the product name attribute and also the whole data hash. The strong parameters API doesn't let you directly whitelist the whole of a nested hash with any keys, but you can use the keys of your nested hash to declare what to whitelist:

```
def product_params
  params.require(:product).permit(:name, data: 
params[:product][:data].try(:keys))
end
```

5 Session

Your application has a session for each user in which you can store small amounts of data that will be persisted between requests. The session is only available in the controller and the view and can use one of a number of different storage mechanisms:

- `ActionDispatch::Session::CookieStore` - Stores everything on the client.
- `ActionDispatch::Session::CacheStore` - Stores the data in the Rails cache.

- `ActionDispatch::Session::ActiveRecordStore` - Stores the data in a database using Active Record. (require `activerecord-session_store` gem).
- `ActionDispatch::Session::MemCacheStore` - Stores the data in a memcached cluster (this is a legacy implementation; consider using CacheStore instead).

All session stores use a cookie to store a unique ID for each session (you must use a cookie, Rails will not allow you to pass the session ID in the URL as this is less secure).

For most stores, this ID is used to look up the session data on the server, e.g. in a database table. There is one exception, and that is the default and recommended session store - the CookieStore - which stores all session data in the cookie itself (the ID is still available to you if you need it). This has the advantage of being very lightweight and it requires zero setup in a new application in order to use the session. The cookie data is cryptographically signed to make it tamper-proof. And it is also encrypted so anyone with access to it can't read its contents. (Rails will not accept it if it has been edited).

The CookieStore can store around 4kB of data - much less than the others - but this is usually enough. Storing large amounts of data in the session is discouraged no matter which session store your application uses. You should especially avoid storing complex objects (anything other than basic Ruby objects, the most common example being model instances) in the session, as the server might not be able to reassemble them between requests, which will result in an error.

If your user sessions don't store critical data or don't need to be around for long periods (for instance if you just use the flash for messaging), you can consider using `ActionDispatch::Session::CacheStore`. This will store sessions using the cache implementation you have configured for your application. The advantage of this is that you can use your existing cache infrastructure for storing sessions without requiring any additional setup or administration. The downside, of course, is that the sessions will be ephemeral and could disappear at any time.

Read more about session storage in the Security Guide.

If you need a different session storage mechanism, you can change it in an initializer:

```
# Use the database for sessions instead of the cookie-
based default,
# which shouldn't be used to store highly confidential
information
# (create the session table with "rails g
active_record:session_migration")
#
Rails.application.config.session_store :active_record_st
ore
```
Rails sets up a session key (the name of the cookie) when signing the session data. These can also be changed in an initializer:
```
# Be sure to restart your server when you modify this
file.
Rails.application.config.session_store :cookie_store,
key: '_your_app_session'
```
You can also pass a `:domain` key and specify the domain name for the cookie:
```
# Be sure to restart your server when you modify this
file.
Rails.application.config.session_store :cookie_store,
key: '_your_app_session', domain: ".example.com"
```
Rails sets up (for the CookieStore) a secret key used for signing the session data in `config/credentials.yml.enc`. This can be changed with `bin/rails credentials:edit`.
```
# aws:
#   access_key_id: 123
#   secret_access_key: 345

# Used as the base secret for all MessageVerifiers in
Rails, including the one protecting cookies.
secret_key_base: 492f...
```
Changing the secret_key_base when using the `CookieStore` will invalidate all existing sessions.

5.1 Accessing the Session

In your controller you can access the session through the `session` instance method.

Sessions are lazily loaded. If you don't access sessions in your action's code, they will not be loaded. Hence you will never need to disable sessions, just not accessing them will do the job.

Session values are stored using key/value pairs like a hash:

```
class ApplicationController < ActionController::Base

  private

  # Finds the User with the ID stored in the session 
with the key
  # :current_user_id This is a common way to handle user 
login in
  # a Rails application; logging in sets the session 
value and
  # logging out removes it.
  def current_user
    @_current_user ||= session[:current_user_id] &&
      User.find_by(id: session[:current_user_id])
  end
end
```
To store something in the session, just assign it to the key like a hash:
```
class LoginsController < ApplicationController
  # "Create" a login, aka "log the user in"
  def create
    if user = User.authenticate(params[:username], 
params[:password])
      # Save the user ID in the session so it can be 
used in
      # subsequent requests
      session[:current_user_id] = user.id
      redirect_to root_url
    end
  end
end
```
To remove something from the session, assign that key to be `nil`:
```
class LoginsController < ApplicationController
  # "Delete" a login, aka "log the user out"
  def destroy
    # Remove the user id from the session
    @_current_user = session[:current_user_id] = nil
    redirect_to root_url
  end
end
```
To reset the entire session, use `reset_session`.

5.2 The Flash

The flash is a special part of the session which is cleared with each request. This means that values stored there will only be available in the next request, which is useful for passing error messages etc.

It is accessed in much the same way as the session, as a hash (it's a FlashHash instance).
Let's use the act of logging out as an example. The controller can send a message which will be displayed to the user on the next request:

```
class LoginsController < ApplicationController
  def destroy
    session[:current_user_id] = nil
    flash[:notice] = "You have successfully logged out."
    redirect_to root_url
  end
end
```

Note that it is also possible to assign a flash message as part of the redirection. You can assign `:notice`, `:alert` or the general purpose `:flash`:

```
redirect_to root_url, notice: "You have successfully
logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

The `destroy` action redirects to the application's `root_url`, where the message will be displayed. Note that it's entirely up to the next action to decide what, if anything, it will do with what the previous action put in the flash. It's conventional to display any error alerts or notices from the flash in the application's layout:

```
<html>
  <!-- <head/> -->
  <body>
    <% flash.each do |name, msg| -%>
      <%= content_tag :div, msg, class: name %>
    <% end -%>

    <!-- more content -->
  </body>
</html>
```

This way, if an action sets a notice or an alert message, the layout will display it automatically.
You can pass anything that the session can store; you're not limited to notices and alerts:

```
<% if flash[:just_signed_up] %>
  <p class="welcome">Welcome to our site!</p>
<% end %>
```

If you want a flash value to be carried over to another request, use the `keep` method:

```
class MainController < ApplicationController
  # Let's say this action corresponds to root_url, but 
you want
  # all requests here to be redirected to 
UsersController#index.
  # If an action sets the flash and redirects here, the 
values
  # would normally be lost when another redirect 
happens, but you
  # can use 'keep' to make it persist for another 
request.
  def index
    # Will persist all flash values.
    flash.keep

    # You can also use a key to keep only some kind of 
value.
    # flash.keep(:notice)
    redirect_to users_url
  end
end
```

5.2.1 `flash.now`

By default, adding values to the flash will make them available to the next request, but sometimes you may want to access those values in the same request. For example, if the `create` action fails to save a resource and you render the `new` template directly, that's not going to result in a new request, but you may still want to display a message using the flash. To do this, you can use `flash.now` in the same way you use the normal `flash`:

```
class ClientsController < ApplicationController
  def create
    @client = Client.new(params[:client])
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render action: "new"
    end
  end
end
```

6 Cookies

Your application can store small amounts of data on the client - called cookies - that will be persisted across requests and even sessions. Rails provides easy access to cookies via the `cookies` method, which - much like the `session` - works like a hash:

```
class CommentsController < ApplicationController
  def new
    # Auto-fill the commenter's name if it has been 
stored in a cookie
    @comment = Comment.new(author: 
cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(params[:comment])
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.author
      else
        # Delete cookie for the commenter's name cookie, 
if any.
        cookies.delete(:commenter_name)
      end
      redirect_to @comment.article
    else
      render action: "new"
    end
  end
end
```

Note that while for session values you set the key to `nil`, to delete a cookie value you should use `cookies.delete(:key)`.
Rails also provides a signed cookie jar and an encrypted cookie jar for storing sensitive data. The signed cookie jar appends a cryptographic signature on the cookie values to protect their integrity. The encrypted cookie jar encrypts the values in addition to signing them, so that they cannot be read by the end user. Refer to the API documentation for more details.
These special cookie jars use a serializer to serialize the assigned values into strings and deserializes them into Ruby objects on read.
You can specify what serializer to use:

```
Rails.application.config.action_dispatch.cookies_seriali
zer = :json
```

The default serializer for new applications is `:json`. For compatibility with old applications with existing cookies, `:marshal` is used when `serializer` option is not specified.

You may also set this option to `:hybrid`, in which case Rails would transparently deserialize existing (`Marshal`-serialized) cookies on read and re-write them in the `JSON` format. This is useful for migrating existing applications to the `:json` serializer.
It is also possible to pass a custom serializer that responds to `load` and `dump`:

```
Rails.application.config.action_dispatch.cookies_seriali
zer = MyCustomSerializer
```

When using the `:json` or `:hybrid` serializer, you should beware that not all Ruby objects can be serialized as JSON. For example, `Date` and `Time` objects will be serialized as strings, and `Hashes` will have their keys stringified.

```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow 
# => Thu, 20 Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => 
"2014-03-20"
  end
end
```

It's advisable that you only store simple data (strings and numbers) in cookies. If you have to store complex objects, you would need to handle the conversion manually when reading the values on subsequent requests.
If you use the cookie session store, this would apply to the `session` and `flash` hash as well.

7 Rendering XML and JSON data

ActionController makes it extremely easy to render `XML` or `JSON` data. If you've generated a controller using scaffolding, it would look something like this:

```
class UsersController < ApplicationController
  def index
    @users = User.all
    respond_to do |format|
      format.html # index.html.erb
      format.xml  { render xml: @users }
      format.json { render json: @users }
    end
  end
end
```

You may notice in the above code that we're using `render xml: @users`, not `render xml: @users.to_xml`. If the object is not a String, then Rails will automatically invoke `to_xml` for us.

8 Filters

Filters are methods that are run "before", "after" or "around" a controller action.

Filters are inherited, so if you set a filter on `ApplicationController`, it will be run on every controller in your application.

"before" filters may halt the request cycle. A common "before" filter is one which requires that a user is logged in for an action to be run. You can define the filter method this way:

```
class ApplicationController < ActionController::Base
  before_action :require_login

  private

  def require_login
    unless logged_in?
      flash[:error] = "You must be logged in to access 
this section"
      redirect_to new_login_url # halts request cycle
    end
  end
end
```

The method simply stores an error message in the flash and redirects to the login form if the user is not logged in. If a "before" filter renders or redirects, the action will not run. If there are additional filters scheduled to run after that filter, they are also cancelled.

In this example the filter is added to `ApplicationController` and thus all controllers in the application inherit it. This will make everything in the application require the user to be logged in in order to use it. For obvious reasons (the user wouldn't be able to log in in the first place!), not all controllers or actions should require this. You can prevent this filter from running before particular actions with `skip_before_action`:

```
class LoginsController < ApplicationController
  skip_before_action :require_login, only: 
[:new, :create]
end
```

Now, the `LoginsController`'s `new` and `create` actions will work as before without requiring the user to be logged in. The `:only` option is used to skip this filter only for these actions, and there is also an `:except` option which works the other way. These options can be used when adding filters too, so you can add a filter which only runs for selected actions in the first place.

Calling the same filter multiple times with different options will not work, since the last filter definition will overwrite the previous ones.

8.1 After Filters and Around Filters

In addition to "before" filters, you can also run filters after an action has been executed, or both before and after.

"after" filters are similar to "before" filters, but because the action has already been run they have access to the response data that's about to be sent to the client. Obviously, "after" filters cannot stop the action from running. Please note that "after" filters are executed only after a successful action, but not when an exception is raised in the request cycle.

"around" filters are responsible for running their associated actions by yielding, similar to how Rack middlewares work.

For example, in a website where changes have an approval workflow an administrator could be able to preview them easily, just apply them within a transaction:

```
class ChangesController < ApplicationController
  around_action :wrap_in_transaction, only: :show

  private

  def wrap_in_transaction
    ActiveRecord::Base.transaction do
      begin
        yield
      ensure
        raise ActiveRecord::Rollback
      end
    end
  end
end
```

Note that an "around" filter also wraps rendering. In particular, if in the example above, the view itself reads from the database (e.g. via a scope), it will do so within the transaction and thus present the data to preview.
You can choose not to yield and build the response yourself, in which case the action will not be run.

8.2 Other Ways to Use Filters

While the most common way to use filters is by creating private methods and using *_action to add them, there are two other ways to do the same thing.
The first is to use a block directly with the *_action methods. The block receives the controller as an argument. The `require_login` filter from above could be rewritten to use a block:

```
class ApplicationController < ActionController::Base
  before_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access 
this section"
      redirect_to new_login_url
    end
  end
end
```

Note that the filter in this case uses `send` because the `logged_in?` method is private and the filter does not run in the scope of the controller. This is not the recommended way to implement this particular filter, but in more simple cases it might be useful.
The second way is to use a class (actually, any object that responds to the right methods will do) to handle the filtering. This is useful in cases that are more complex and cannot be implemented in a readable and reusable way using the two other methods. As an example, you could rewrite the login filter again to use a class:

```
class ApplicationController < ActionController::Base
  before_action LoginFilter
end

class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in 
to access this section"
      controller.redirect_to controller.new_login_url
    end
  end
end
```

Again, this is not an ideal example for this filter, because it's not run in the scope of the controller but gets the controller passed as an argument. The filter class must implement a method with the same name as the filter, so for the `before_action` filter the class must implement a `before` method, and so on. The `around` method must `yield` to execute the action.

9 Request Forgery Protection

Cross-site request forgery is a type of attack in which a site tricks a user into making requests on another site, possibly adding, modifying or deleting data on that site without the user's knowledge or permission.
The first step to avoid this is to make sure all "destructive" actions (create, update and destroy) can only be accessed with non-GET requests. If you're following RESTful conventions you're already doing this. However, a malicious site can still send a non-GET request to your site quite easily, and that's where the request forgery protection comes in. As the name says, it protects from forged requests.
The way this is done is to add a non-guessable token which is only known to your server to each request. This way, if a request comes in without the proper token, it will be denied access.
If you generate a form like this:

```
<%= form_with model: @user, local: true do |form| %>
  <%= form.text_field :username %>
  <%= form.text_field :password %>
<% end %>
```

You will see how the token gets added as a hidden field:

```
<form accept-charset="UTF-8" action="/users/1"
method="post">
<input type="hidden"
       value="67250ab105eb5ad10851c00a5621854a23af5489"
       name="authenticity_token"/>
<!-- fields -->
</form>
```

Rails adds this token to every form that's generated using the form helpers, so most of the time you don't have to worry about it. If you're writing a form manually or need to add the token for another reason, it's available through the method `form_authenticity_token`:
The `form_authenticity_token` generates a valid authentication token. That's useful in places where Rails does not add it automatically, like in custom Ajax calls.
The Security Guide has more about this and a lot of other security-related issues that you should be aware of when developing a web application.

10 The Request and Response Objects

In every controller there are two accessor methods pointing to the request and the response objects associated with the request cycle that is currently in execution. The `request` method contains an instance of `ActionDispatch::Request` and the `response` method returns a response object representing what is going to be sent back to the client.

10.1 The `request` Object

The request object contains a lot of useful information about the request coming in from the client. To get a full list of the available methods, refer to the Rails API documentation and Rack Documentation. Among the properties that you can access on this object are:

Property of `request`	Purpose
host	The hostname used for this request.
domain(n=2)	The hostname's first `n` segments, starting from the right (the TLD).
format	The content type requested by the client.
method	The HTTP method used for the request.
get?, post?, patch?, put?, delete?, head?	Returns true if the HTTP method is GET/POST/PATCH/PUT/DELETE/HEAD.
headers	Returns a hash containing the headers associated with the request.
port	The port number (integer) used for the request.
protocol	Returns a string containing the protocol used plus "://", for example "http://".
query_string	The query string part of the URL, i.e., everything after "?".
remote_ip	The IP address of the client.

url	The entire URL used for the request.

10.1.1 `path_parameters`, `query_parameters`, and `request_parameters`

Rails collects all of the parameters sent along with the request in the `params` hash, whether they are sent as part of the query string or the post body. The request object has three accessors that give you access to these parameters depending on where they came from. The `query_parameters` hash contains parameters that were sent as part of the query string while the `request_parameters` hash contains parameters sent as part of the post body. The `path_parameters` hash contains parameters that were recognized by the routing as being part of the path leading to this particular controller and action.

10.2 The `response` Object

The response object is not usually used directly, but is built up during the execution of the action and rendering of the data that is being sent back to the user, but sometimes - like in an after filter - it can be useful to access the response directly. Some of these accessor methods also have setters, allowing you to change their values. To get a full list of the available methods, refer to the Rails API documentation and Rack Documentation.

Property of `response`	Purpose
body	This is the string of data being sent back to the client. This is most often HTML.
status	The HTTP status code for the response, like 200 for a successful request or 404 for file not found.
location	The URL the client is being redirected to, if any.
content_type	The content type of the response.
charset	The character set being used for the response. Default is "utf-8".
headers	Headers used for the response.

10.2.1 Setting Custom Headers

If you want to set custom headers for a response then `response.headers` is the place to do it. The headers attribute is a hash which maps header names to their values, and Rails will set some of them automatically. If you want to add or change a header, just assign it to `response.headers` this way:

```
response.headers["Content-Type"] = "application/pdf"
```

Note: in the above case it would make more sense to use the `content_type` setter directly.

11 HTTP Authentications

Rails comes with two built-in HTTP authentication mechanisms:

- Basic Authentication
- Digest Authentication

11.1 HTTP Basic Authentication

HTTP basic authentication is an authentication scheme that is supported by the majority of browsers and other HTTP clients. As an example, consider an administration section which will only be available by entering a username and a password into the browser's HTTP basic dialog window. Using the built-in authentication is quite easy and only requires you to use one method, `http_basic_authenticate_with`.

```
class AdminsController < ApplicationController
  http_basic_authenticate_with name: "humbaba", 
password: "5baa61e4"
end
```

With this in place, you can create namespaced controllers that inherit from `AdminsController`. The filter will thus be run for all actions in those controllers, protecting them with HTTP basic authentication.

11.2 HTTP Digest Authentication

HTTP digest authentication is superior to the basic authentication as it does not require the client to send an unencrypted password over the network (though HTTP basic authentication is safe over HTTPS). Using digest authentication with Rails is quite easy and only requires using one method, `authenticate_or_request_with_http_digest`.

```
class AdminsController < ApplicationController
  USERS = { "lifo" => "world" }

  before_action :authenticate

  private

    def authenticate
      authenticate_or_request_with_http_digest do |
username|
        USERS[username]
      end
    end
end
```

As seen in the example above, the `authenticate_or_request_with_http_digest` block takes only one argument - the username. And the block returns the password. Returning `false` or `nil` from the `authenticate_or_request_with_http_digest` will cause authentication failure.

12 Streaming and File Downloads

Sometimes you may want to send a file to the user instead of rendering an HTML page. All controllers in Rails have the `send_data` and the `send_file` methods, which will both stream data to the client. `send_file` is a convenience method that lets you provide the name of a file on the disk and it will stream the contents of that file for you.

To stream data to the client, use `send_data`:

```
require "prawn"
class ClientsController < ApplicationController
  # Generates a PDF document with information on the 
client and
  # returns it. The user will get the PDF as a file 
download.
  def download_pdf
    client = Client.find(params[:id])
    send_data generate_pdf(client),
              filename: "#{client.name}.pdf",
              type: "application/pdf"
  end

  private

    def generate_pdf(client)
      Prawn::Document.new do
        text client.name, align: :center
        text "Address: #{client.address}"
        text "Email: #{client.email}"
      end.render
    end
end
```

The `download_pdf` action in the example above will call a private method which actually generates the PDF document and returns it as a string. This string will then be streamed to the client as a file download and a filename will be suggested to the user. Sometimes when streaming files to the user, you may not want them to download the file. Take images, for example, which can be embedded into HTML pages. To tell the browser a file is not meant to be downloaded, you can set the `:disposition` option to "inline". The opposite and default value for this option is "attachment".

12.1 Sending Files

If you want to send a file that already exists on disk, use the `send_file` method.

```
class ClientsController < ApplicationController
  # Stream a file that has already been generated and 
stored on disk.
  def download_pdf
    client = Client.find(params[:id])
    send_file("#{Rails.root}/files/clients/
#{client.id}.pdf",
              filename: "#{client.name}.pdf",
              type: "application/pdf")
  end
end
```

This will read and stream the file 4kB at the time, avoiding loading the entire file into memory at once. You can turn off streaming with the `:stream` option or adjust the block size with the `:buffer_size` option.

If `:type` is not specified, it will be guessed from the file extension specified in `:filename`. If the content type is not registered for the extension, `application/octet-stream` will be used.

Be careful when using data coming from the client (params, cookies, etc.) to locate the file on disk, as this is a security risk that might allow someone to gain access to files they are not meant to.

It is not recommended that you stream static files through Rails if you can instead keep them in a public folder on your web server. It is much more efficient to let the user download the file directly using Apache or another web server, keeping the request from unnecessarily going through the whole Rails stack.

12.2 RESTful Downloads

While `send_data` works just fine, if you are creating a RESTful application having separate actions for file downloads is usually not necessary. In REST terminology, the PDF file from the example above can be considered just another representation of the client resource. Rails provides an easy and quite sleek way of doing "RESTful downloads". Here's how you can rewrite the example so that the PDF download is a part of the `show` action, without any streaming:

```
class ClientsController < ApplicationController
  # The user can request to receive this resource as
HTML or PDF.
  def show
    @client = Client.find(params[:id])

    respond_to do |format|
      format.html
      format.pdf { render pdf: generate_pdf(@client) }
    end
  end
end
```

In order for this example to work, you have to add the PDF MIME type to Rails. This can be done by adding the following line to the file `config/initializers/mime_types.rb`:

```
Mime::Type.register "application/pdf", :pdf
```

Configuration files are not reloaded on each request, so you have to restart the server in order for their changes to take effect.

Now the user can request to get a PDF version of a client just by adding ".pdf" to the URL:

```
GET /clients/1.pdf
```

12.3 Live Streaming of Arbitrary Data

Rails allows you to stream more than just files. In fact, you can stream anything you would like in a response object. The `ActionController::Live` module allows you to create a persistent connection with a browser. Using this module, you will be able to send arbitrary data to the browser at specific points in time.

12.3.1 Incorporating Live Streaming

Including `ActionController::Live` inside of your controller class will provide all actions inside of the controller the ability to stream data. You can mix in the module like so:

```
class MyController < ActionController::Base
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-
stream'
    100.times {
      response.stream.write "hello world\n"
      sleep 1
    }
  ensure
    response.stream.close
  end
end
```

The above code will keep a persistent connection with the browser and send 100 messages of `"hello world\n"`, each one second apart.

There are a couple of things to notice in the above example. We need to make sure to close the response stream. Forgetting to close the stream will leave the socket open forever. We also have to set the content type to `text/event-stream` before we write to the response stream. This is because headers cannot be written after the response has been committed (when `response.committed?` returns a truthy value), which occurs when you `write` or `commit` the response stream.

12.3.2 Example Usage

Let's suppose that you were making a Karaoke machine and a user wants to get the lyrics for a particular song. Each `Song` has a particular number of lines and each line takes time `num_beats` to finish singing.

If we wanted to return the lyrics in Karaoke fashion (only sending the line when the singer has finished the previous line), then we could use `ActionController::Live` as follows:

```
class LyricsController < ActionController::Base
  include ActionController::Live

  def show
    response.headers['Content-Type'] = 'text/event-
stream'
    song = Song.find(params[:id])

    song.each do |line|
      response.stream.write line.lyrics
      sleep line.num_beats
    end
  ensure
    response.stream.close
  end
end
```

The above code sends the next line only after the singer has completed the previous line.

12.3.3 Streaming Considerations

Streaming arbitrary data is an extremely powerful tool. As shown in the previous examples, you can choose when and what to send across a response stream. However, you should also note the following things:

- Each response stream creates a new thread and copies over the thread local variables from the original thread. Having too many thread local variables can negatively impact performance. Similarly, a large number of threads can also hinder performance.
- Failing to close the response stream will leave the corresponding socket open forever. Make sure to call `close` whenever you are using a response stream.
- WEBrick servers buffer all responses, and so including `ActionController::Live` will not work. You must use a web server which does not automatically buffer responses.

13 Log Filtering

Rails keeps a log file for each environment in the `log` folder. These are extremely useful when debugging what's actually going on in your application, but in a live application you may not want every bit of information to be stored in the log file.

13.1 Parameters Filtering

You can filter out sensitive request parameters from your log files by appending them to `config.filter_parameters` in the application configuration. These parameters will be marked [FILTERED] in the log.

```
config.filter_parameters << :password
```

Provided parameters will be filtered out by partial matching regular expression. Rails adds default `:password` in the appropriate initializer (`initializers/filter_parameter_logging.rb`) and cares about typical application parameters `password` and `password_confirmation`.

13.2 Redirects Filtering

Sometimes it's desirable to filter out from log files some sensitive locations your application is redirecting to. You can do that by using the `config.filter_redirect` configuration option:

```
config.filter_redirect << 's3.amazonaws.com'
```

You can set it to a String, a Regexp, or an array of both.

```
config.filter_redirect.concat ['s3.amazonaws.com', /
private_path/]
```

Matching URLs will be marked as '[FILTERED]'.

14 Rescue

Most likely your application is going to contain bugs or otherwise throw an exception that needs to be handled. For example, if the user follows a link to a resource that no longer exists in the database, Active Record will throw the `ActiveRecord::RecordNotFound` exception.

Rails default exception handling displays a "500 Server Error" message for all exceptions. If the request was made locally, a nice traceback and some added information gets displayed so you can figure out what went wrong and deal with it. If the request was remote Rails will just display a simple "500 Server Error" message to the user, or a "404 Not Found" if there was a routing error or a record could not be found. Sometimes you might want to customize how these errors are caught and how they're displayed to the user. There are several levels of exception handling available in a Rails application:

14.1 The Default 500 and 404 Templates

By default a production application will render either a 404 or a 500 error message, in the development environment all unhandled exceptions are raised. These messages are contained in static HTML files in the public folder, in `404.html` and `500.html` respectively. You can customize these files to add some extra information and style, but remember that they are static HTML; i.e. you can't use ERB, SCSS, CoffeeScript, or layouts for them.

14.2 `rescue_from`

If you want to do something a bit more elaborate when catching errors, you can use `rescue_from`, which handles exceptions of a certain type (or multiple types) in an entire controller and its subclasses.

When an exception occurs which is caught by a `rescue_from` directive, the exception object is passed to the handler. The handler can be a method or a

`Proc` object passed to the `:with` option. You can also use a block directly instead of an explicit `Proc` object.
Here's how you can use `rescue_from` to intercept all `ActiveRecord::RecordNotFound` errors and do something with them.

```
class ApplicationController < ActionController::Base
  rescue_from ActiveRecord::RecordNotFound,
with: :record_not_found

  private

    def record_not_found
      render plain: "404 Not Found", status: 404
    end
end
```

Of course, this example is anything but elaborate and doesn't improve on the default exception handling at all, but once you can catch all those exceptions you're free to do whatever you want with them. For example, you could create custom exception classes that will be thrown when a user doesn't have access to a certain section of your application:

```
class ApplicationController < ActionController::Base
  rescue_from User::NotAuthorized,
with: :user_not_authorized

  private

    def user_not_authorized
      flash[:error] = "You don't have access to this
section."
      redirect_back(fallback_location: root_path)
    end
end

class ClientsController < ApplicationController
  # Check that the user has the right authorization to
access clients.
  before_action :check_authorization

  # Note how the actions don't have to worry about all
the auth stuff.
  def edit
    @client = Client.find(params[:id])
  end

  private

    # If the user is not authorized, just throw the
exception.
    def check_authorization
      raise User::NotAuthorized unless
current_user.admin?
    end
end
```

Using `rescue_from` with `Exception` or `StandardError` would cause serious side-effects as it prevents Rails from handling exceptions properly. As such, it is not recommended to do so unless there is a strong reason.

When running in the production environment, all `ActiveRecord::RecordNotFound` errors render the 404 error page. Unless you need a custom behavior you don't need to handle this.

Certain exceptions are only rescuable from the `ApplicationController` class, as they are raised before the controller gets initialized and the action gets executed.

15 Force HTTPS protocol

Sometime you might want to force a particular controller to only be accessible via an HTTPS protocol for security reasons. You can use the `force_ssl` method in your controller to enforce that:

```
class DinnerController
  force_ssl
end
```

Just like the filter, you could also pass `:only` and `:except` to enforce the secure connection only to specific actions:

```
class DinnerController
  force_ssl only: :cheeseburger
  # or
  force_ssl except: :cheeseburger
end
```

Please note that if you find yourself adding `force_ssl` to many controllers, you may want to force the whole application to use HTTPS instead. In that case, you can set the `config.force_ssl` in your environment file.

1 Dealing with Basic Forms

The most basic form helper is `form_tag`.

```
<%= form_tag do %>
  Form contents
<% end %>
```

When called without arguments like this, it creates a `<form>` tag which, when submitted, will POST to the current page. For instance, assuming the current page is `/home/index`, the generated HTML will look like this (some line breaks added for readability):

```
<form accept-charset="UTF-8" action="/" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden"
value="J7CBxfHalt49OSHp27hblqK20c9PgwJ108nDHX/8Cts=" />
  Form contents
</form>
```

You'll notice that the HTML contains an `input` element with type `hidden`. This `input` is important, because the form cannot be successfully submitted without it. The hidden input element with the name `utf8` enforces browsers to properly respect your form's character encoding and is generated for all forms whether their action is "GET" or "POST".

The second input element with the name `authenticity_token` is a security feature of Rails called **cross-site request forgery protection**, and form helpers generate it for every non-GET form (provided that this security feature is enabled). You can read more about this in the Security Guide.

1.1 A Generic Search Form

One of the most basic forms you see on the web is a search form. This form contains:

- a form element with "GET" method,
- a label for the input,
- a text input element, and
- a submit element.

To create this form you will use `form_tag`, `label_tag`, `text_field_tag`, and `submit_tag`, respectively. Like this:

```
<%= form_tag("/search", method: "get") do %>
  <%= label_tag(:q, "Search for:") %>
  <%= text_field_tag(:q) %>
  <%= submit_tag("Search") %>
<% end %>
```

This will generate the following HTML:

```
<form accept-charset="UTF-8" action="/search"
method="get">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <label for="q">Search for:</label>
  <input id="q" name="q" type="text" />
  <input name="commit" type="submit" value="Search" />
</form>
```

For every form input, an ID attribute is generated from its name (`"q"` in above example). These IDs can be very useful for CSS styling or manipulation of form controls with JavaScript.

Besides `text_field_tag` and `submit_tag`, there is a similar helper for *every* form control in HTML.

Always use "GET" as the method for search forms. This allows users to bookmark a specific search and get back to it. More generally Rails encourages you to use the right HTTP verb for an action.

1.2 Multiple Hashes in Form Helper Calls

The `form_tag` helper accepts 2 arguments: the path for the action and an options hash. This hash specifies the method of form submission and HTML options such as the form element's class.

As with the `link_to` helper, the path argument doesn't have to be a string; it can be a hash of URL parameters recognizable by Rails' routing mechanism, which will turn the hash into a valid URL. However, since both arguments to `form_tag` are hashes, you can easily run into a problem if you would like to specify both. For instance, let's say you write this:

```
form_tag(controller: "people", action: "search", method:
"get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/
search?method=get&class=nifty_form" method="post">'
```

Here, `method` and `class` are appended to the query string of the generated URL because even though you mean to write two hashes, you really only specified one. So you need to tell Ruby which is which by delimiting the first hash (or both) with curly brackets. This will generate the HTML you expect:

```
form_tag({controller: "people", action: "search"},
method: "get", class: "nifty_form")
# => '<form accept-charset="UTF-8" action="/people/
search" method="get" class="nifty_form">'
```

1.3 Helpers for Generating Form Elements

Rails provides a series of helpers for generating form elements such as checkboxes, text fields, and radio buttons. These basic helpers, with names ending in `_tag` (such as `text_field_tag` and `check_box_tag`), generate just a single `<input>` element. The first parameter to these is always the name of the input. When the form is submitted, the name will be passed

along with the form data, and will make its way to the `params` in the controller with the value entered by the user for that field. For example, if the form contains `<%= text_field_tag(:query) %>`, then you would be able to get the value of this field in the controller with `params[:query]`.
When naming inputs, Rails uses certain conventions that make it possible to submit parameters with non-scalar values such as arrays or hashes, which will also be accessible in `params`. You can read more about them in chapter 7 of this guide. For details on the precise usage of these helpers, please refer to the API documentation.

1.3.1 Checkboxes

Checkboxes are form controls that give the user a set of options they can enable or disable:

```
<%= check_box_tag(:pet_dog) %>
<%= label_tag(:pet_dog, "I own a dog") %>
<%= check_box_tag(:pet_cat) %>
<%= label_tag(:pet_cat, "I own a cat") %>
```

This generates the following:

```
<input id="pet_dog" name="pet_dog" type="checkbox"
value="1" />
<label for="pet_dog">I own a dog</label>
<input id="pet_cat" name="pet_cat" type="checkbox"
value="1" />
<label for="pet_cat">I own a cat</label>
```

The first parameter to `check_box_tag`, of course, is the name of the input. The second parameter, naturally, is the value of the input. This value will be included in the form data (and be present in `params`) when the checkbox is checked.

1.3.2 Radio Buttons

Radio buttons, while similar to checkboxes, are controls that specify a set of options in which they are mutually exclusive (i.e., the user can only pick one):

```
<%= radio_button_tag(:age, "child") %>
<%= label_tag(:age_child, "I am younger than 21") %>
<%= radio_button_tag(:age, "adult") %>
<%= label_tag(:age_adult, "I'm over 21") %>
```

Output:

```
<input id="age_child" name="age" type="radio"
value="child" />
<label for="age_child">I am younger than 21</label>
<input id="age_adult" name="age" type="radio"
value="adult" />
<label for="age_adult">I'm over 21</label>
```

As with `check_box_tag`, the second parameter to `radio_button_tag` is the value of the input. Because these two radio buttons share the same name (`age`), the user will only be able to select one of them, and `params[:age]` will contain either `"child"` or `"adult"`.

Always use labels for checkbox and radio buttons. They associate text with a specific option and, by expanding the clickable region, make it easier for users to click the inputs.

1.4 Other Helpers of Interest

Other form controls worth mentioning are textareas, password fields, hidden fields, search fields, telephone fields, date fields, time fields, color fields, datetime-local fields, month fields, week fields, URL fields, email fields, number fields and range fields:

```
<%= text_area_tag(:message, "Hi, nice site", size:
"24x6") %>
<%= password_field_tag(:password) %>
<%= hidden_field_tag(:parent_id, "5") %>
<%= search_field(:user, :name) %>
<%= telephone_field(:user, :phone) %>
<%= date_field(:user, :born_on) %>
<%= datetime_local_field(:user, :graduation_day) %>
<%= month_field(:user, :birthday_month) %>
<%= week_field(:user, :birthday_week) %>
<%= url_field(:user, :homepage) %>
<%= email_field(:user, :address) %>
<%= color_field(:user, :favorite_color) %>
<%= time_field(:task, :started_at) %>
<%= number_field(:product, :price, in: 1.0..20.0, step:
0.5) %>
<%= range_field(:product, :discount, in: 1..100) %>
```

Output:

```
<textarea id="message" name="message" cols="24"
rows="6">Hi, nice site</textarea>
<input id="password" name="password" type="password" />
<input id="parent_id" name="parent_id" type="hidden"
value="5" />
<input id="user_name" name="user[name]" type="search" />
<input id="user_phone" name="user[phone]" type="tel" />
<input id="user_born_on" name="user[born_on]"
type="date" />
<input id="user_graduation_day"
name="user[graduation_day]" type="datetime-local" />
<input id="user_birthday_month"
name="user[birthday_month]" type="month" />
<input id="user_birthday_week"
name="user[birthday_week]" type="week" />
<input id="user_homepage" name="user[homepage]"
type="url" />
<input id="user_address" name="user[address]"
type="email" />
<input id="user_favorite_color"
name="user[favorite_color]" type="color" value="#000000"
/>
<input id="task_started_at" name="task[started_at]"
type="time" />
<input id="product_price" max="20.0" min="1.0"
name="product[price]" step="0.5" type="number" />
<input id="product_discount" max="100" min="1"
name="product[discount]" type="range" />
```

Hidden inputs are not shown to the user but instead hold data like any textual input. Values inside them can be changed with JavaScript.

The search, telephone, date, time, color, datetime, datetime-local, month, week, URL, email, number and range inputs are HTML5 controls. If you require your app to have a consistent experience in older browsers, you will need an HTML5 polyfill (provided by CSS and/or JavaScript). There is definitely no shortage of solutions for this, although a popular tool at the moment is Modernizr, which provides a simple way to add functionality based on the presence of detected HTML5 features.

If you're using password input fields (for any purpose), you might want to configure your application to prevent those parameters from being logged. You can learn about this in the Security Guide.

2 Dealing with Model Objects

2.1 Model Object Helpers

A particularly common task for a form is editing or creating a model object. While the `*_tag` helpers can certainly be used for this task they are somewhat verbose as for each tag you would have to ensure the correct parameter name is used and set the default value of the input appropriately. Rails provides helpers tailored to this task. These helpers lack the `_tag` suffix, for example `text_field`, `text_area`.

For these helpers the first argument is the name of an instance variable and the second is the name of a method (usually an attribute) to call on that object. Rails will set the value of the input control to the return value of that method for the object and set an appropriate input name. If your controller has defined `@person` and that person's name is Henry then a form containing:

```
<%= text_field(:person, :name) %>
```

will produce output similar to

```
<input id="person_name" name="person[name]" type="text"
value="Henry"/>
```

Upon form submission the value entered by the user will be stored in `params[:person][:name]`. The `params[:person]` hash is suitable for passing to `Person.new` or, if `@person` is an instance of Person, `@person.update`. While the name of an attribute is the most common second parameter to these helpers this is not compulsory. In the example above, as long as person objects have a `name` and a `name=` method Rails will be happy.

You must pass the name of an instance variable, i.e. `:person` or `"person"`, not an actual instance of your model object.

Rails provides helpers for displaying the validation errors associated with a model object. These are covered in detail by the Active Record Validations guide.

2.2 Binding a Form to an Object

While this is an increase in comfort it is far from perfect. If `Person` has many attributes to edit then we would be repeating the name of the edited object many times. What we want to do is somehow bind a form to a model object, which is exactly what `form_for` does.

Assume we have a controller for dealing with articles `app/controllers/articles_controller.rb`:

```
def new
  @article = Article.new
end
```
The corresponding view `app/views/articles/new.html.erb` using `form_for` looks like this:
```
<%= form_for @article, url: {action: "create"}, html:
{class: "nifty_form"} do |f| %>
  <%= f.text_field :title %>
  <%= f.text_area :body, size: "60x12" %>
  <%= f.submit "Create" %>
<% end %>
```
There are a few things to note here:

- `@article` is the actual object being edited.
- There is a single hash of options. Routing options are passed in the `:url` hash, HTML options are passed in the `:html` hash. Also you can provide a `:namespace` option for your form to ensure uniqueness of id attributes on form elements. The namespace attribute will be prefixed with underscore on the generated HTML id.
- The `form_for` method yields a **form builder** object (the `f` variable).
- Methods to create form controls are called **on** the form builder object `f`.

The resulting HTML is:
```
<form class="nifty_form" id="new_article" action="/
articles" accept-charset="UTF-8" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input type="hidden" name="authenticity_token"
value="NRkFyRWxdYNfUg7vYxLOp2SLf93lvnl+QwDWorR42Dp6yZXPh
HEb6arhDOIWcqGit8jfnrPwL781/xlrzj63TA==" />
  <input type="text" name="article[title]"
id="article_title" />
  <textarea name="article[body]" id="article_body"
cols="60" rows="12"></textarea>
  <input type="submit" name="commit" value="Create"
data-disable-with="Create" />
</form>
```
The name passed to `form_for` controls the key used in `params` to access the form's values. Here the name is `article` and so all the inputs have names of the form `article[attribute_name]`. Accordingly, in the `create` action `params[:article]` will be a hash with keys `:title` and `:body`. You can read more about the significance of input names in the parameter_names section.

The helper methods called on the form builder are identical to the model object helpers except that it is not necessary to specify which object is being edited since this is already managed by the form builder.

You can create a similar binding without actually creating `<form>` tags with the `fields_for` helper. This is useful for editing additional model objects with the same form. For example, if you had a `Person` model with an associated `ContactDetail` model, you could create a form for creating both like so:

```
<%= form_for @person, url: {action: "create"} do |
person_form| %>
  <%= person_form.text_field :name %>
  <%= fields_for @person.contact_detail do |
contact_detail_form| %>
    <%= contact_detail_form.text_field :phone_number %>
  <% end %>
<% end %>
```

which produces the following output:

```
<form class="new_person" id="new_person" action="/
people" accept-charset="UTF-8" method="post">
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input type="hidden" name="authenticity_token"
value="bL13x72pldyDD8bgtkjKQakJCpd4A8JdXGbfksxBDHdf1uC0k
CMqe2tvVdUYfidJt0fj3ihC4NxiVHv8GVYxJA==" />
  <input type="text" name="person[name]"
id="person_name" />
  <input type="text" name="contact_detail[phone_number]"
id="contact_detail_phone_number" />
</form>
```

The object yielded by `fields_for` is a form builder like the one yielded by `form_for` (in fact `form_for` calls `fields_for` internally).

2.3 Relying on Record Identification

The Article model is directly available to users of the application, so - following the best practices for developing with Rails - you should declare it **a resource**:

```
resources :articles
```

Declaring a resource has a number of side effects. See Rails Routing From the Outside In for more information on setting up and using resources.

When dealing with RESTful resources, calls to `form_for` can get significantly easier if you rely on **record identification**. In short, you can just pass the model instance and have Rails figure out model name and the rest:

```
## Creating a new article
# long-style:
form_for(@article, url: articles_path)
# same thing, short-style (record identification gets 
used):
form_for(@article)

## Editing an existing article
# long-style:
form_for(@article, url: article_path(@article), html: 
{method: "patch"})
# short-style:
form_for(@article)
```

Notice how the short-style `form_for` invocation is conveniently the same, regardless of the record being new or existing. Record identification is smart enough to figure out if the record is new by asking `record.new_record?`. It also selects the correct path to submit to and the name based on the class of the object.

Rails will also automatically set the `class` and `id` of the form appropriately: a form creating an article would have `id` and `class` `new_article`. If you were editing the article with id 23, the `class` would be set to `edit_article` and the id to `edit_article_23`. These attributes will be omitted for brevity in the rest of this guide.

When you're using STI (single-table inheritance) with your models, you can't rely on record identification on a subclass if only their parent class is declared a resource. You will have to specify the model name, `:url`, and `:method` explicitly.

2.3.1 Dealing with Namespaces

If you have created namespaced routes, `form_for` has a nifty shorthand for that too. If your application has an admin namespace then

```
form_for [:admin, @article]
```

will create a form that submits to the `ArticlesController` inside the admin namespace (submitting to `admin_article_path(@article)` in the case of an update). If you have several levels of namespacing then the syntax is similar:

```
form_for [:admin, :management, @article]
```

For more information on Rails' routing system and the associated conventions, please see the routing guide.

2.4 How do forms with PATCH, PUT, or DELETE methods work?

The Rails framework encourages RESTful design of your applications, which means you'll be making a lot of "PATCH" and "DELETE" requests (besides

"GET" and "POST"). However, most browsers *don't support* methods other than "GET" and "POST" when it comes to submitting forms.
Rails works around this issue by emulating other methods over POST with a hidden input named `"_method"`, which is set to reflect the desired method:

```
form_tag(search_path, method: "patch")
```

output:

```
<form accept-charset="UTF-8" action="/search"
method="post">
  <input name="_method" type="hidden" value="patch" />
  <input name="utf8" type="hidden" value="&#x2713;" />
  <input name="authenticity_token" type="hidden"
value="f755bb0ed134b76c432144748a6d4b7a7ddf2b71" />
  ...
</form>
```

When parsing POSTed data, Rails will take into account the special `_method` parameter and act as if the HTTP method was the one specified inside it ("PATCH" in this example).

3 Making Select Boxes with Ease

Select boxes in HTML require a significant amount of markup (one `OPTION` element for each option to choose from), therefore it makes the most sense for them to be dynamically generated.
Here is what the markup might look like:

```
<select name="city_id" id="city_id">
  <option value="1">Lisbon</option>
  <option value="2">Madrid</option>
  ...
  <option value="12">Berlin</option>
</select>
```

Here you have a list of cities whose names are presented to the user. Internally the application only wants to handle their IDs so they are used as the options' value attribute. Let's see how Rails can help out here.

3.1 The Select and Option Tags

The most generic helper is `select_tag`, which - as the name implies - simply generates the `SELECT` tag that encapsulates an options string:

```
<%= select_tag(:city_id, '<option value="1">Lisbon</
option>...') %>
```

This is a start, but it doesn't dynamically create the option tags. You can generate option tags with the `options_for_select` helper:

```
<%= options_for_select([['Lisbon', 1], ['Madrid',
2], ...]) %>

output:

<option value="1">Lisbon</option>
<option value="2">Madrid</option>
...
```

The first argument to `options_for_select` is a nested array where each element has two elements: option text (city name) and option value (city id). The option value is what will be submitted to your controller. Often this will be the id of a corresponding database object but this does not have to be the case.

Knowing this, you can combine `select_tag` and `options_for_select` to achieve the desired, complete markup:

```
<%= select_tag(:city_id, options_for_select(...)) %>
```

`options_for_select` allows you to pre-select an option by passing its value.

```
<%= options_for_select([['Lisbon', 1], ['Madrid',
2], ...], 2) %>

output:

<option value="1">Lisbon</option>
<option value="2" selected="selected">Madrid</option>
...
```

Whenever Rails sees that the internal value of an option being generated matches this value, it will add the `selected` attribute to that option.

When `:include_blank` or `:prompt` are not present, `:include_blank` is forced true if the select attribute `required` is true, display `size` is one and `multiple` is not true.

You can add arbitrary attributes to the options using hashes:

```
<%= options_for_select(
  [
    ['Lisbon', 1, { 'data-size' => '2.8 million' }],
    ['Madrid', 2, { 'data-size' => '3.2 million' }]
  ], 2
) %>

output:

<option value="1" data-size="2.8 million">Lisbon</
option>
<option value="2" selected="selected" data-size="3.2 
million">Madrid</option>
...
```

3.2 Select Boxes for Dealing with Models

In most cases form controls will be tied to a specific database model and as you might expect Rails provides helpers tailored for that purpose. Consistent with other form helpers, when dealing with models you drop the `_tag` suffix from `select_tag`:

```
# controller:
@person = Person.new(city_id: 2)
```

```
# view:
<%= select(:person, :city_id, [['Lisbon', 1], ['Madrid', 
2], ...]) %>
```

Notice that the third parameter, the options array, is the same kind of argument you pass to `options_for_select`. One advantage here is that you don't have to worry about pre-selecting the correct city if the user already has one - Rails will do this for you by reading from the `@person.city_id` attribute.

As with other helpers, if you were to use the `select` helper on a form builder scoped to the `@person` object, the syntax would be:

```
# select on a form builder
<%= f.select(:city_id, ...) %>
```

You can also pass a block to `select` helper:

```
<%= f.select(:city_id) do %>
  <% [['Lisbon', 1], ['Madrid', 2]].each do |c| -%>
    <%= content_tag(:option, c.first, value: c.last) %>
  <% end %>
<% end %>
```

If you are using `select` (or similar helpers such as `collection_select`, `select_tag`) to set a `belongs_to` association you must pass the name of the foreign key (in the example above `city_id`), not the name of association itself. If you specify `city` instead of `city_id` Active Record will raise an error along the lines of `ActiveRecord::AssociationTypeMismatch: City(#17815740) expected, got String(#1138750)` when you pass the `params` hash to `Person.new` or `update`. Another way of looking at this is that form helpers only edit attributes. You should also be aware of the potential security ramifications of allowing users to edit foreign keys directly.

3.3 Option Tags from a Collection of Arbitrary Objects

Generating options tags with `options_for_select` requires that you create an array containing the text and value for each option. But what if you had a `City` model (perhaps an Active Record one) and you wanted to generate option tags from a collection of those objects? One solution would be to make a nested array by iterating over them:

```
<% cities_array = City.all.map { |city| [city.name,
city.id] } %>
<%= options_for_select(cities_array) %>
```

This is a perfectly valid solution, but Rails provides a less verbose alternative: `options_from_collection_for_select`. This helper expects a collection of arbitrary objects and two additional arguments: the names of the methods to read the option **value** and **text** from, respectively:

```
<%=
options_from_collection_for_select(City.all, :id, :name)
%>
```

As the name implies, this only generates option tags. To generate a working select box you would need to use it in conjunction with `select_tag`, just as you would with `options_for_select`. When working with model objects, just as `select` combines `select_tag` and `options_for_select`, `collection_select` combines `select_tag` with `options_from_collection_for_select`.

```
<%= collection_select(:person, :city_id,
City.all, :id, :name) %>
```

As with other helpers, if you were to use the `collection_select` helper on a form builder scoped to the `@person` object, the syntax would be:

```
<%= f.collection_select(:city_id, City.all, :id, :name)
%>
```

To recap, `options_from_collection_for_select` is to `collection_select` what `options_for_select` is to `select`.

Pairs passed to `options_for_select` should have the name first and the id second, however with `options_from_collection_for_select` the first argument is the value method and the second the text method.

3.4 Time Zone and Country Select

To leverage time zone support in Rails, you have to ask your users what time zone they are in. Doing so would require generating select options from a list of pre-defined TimeZone objects using `collection_select`, but you can simply use the `time_zone_select` helper that already wraps this:

```
<%= time_zone_select(:person, :time_zone) %>
```

There is also `time_zone_options_for_select` helper for a more manual (therefore more customizable) way of doing this. Read the API documentation to learn about the possible arguments for these two methods.

Rails *used* to have a `country_select` helper for choosing countries, but this has been extracted to the country_select plugin. When using this, be aware that the exclusion or inclusion of certain names from the list can be somewhat controversial (and was the reason this functionality was extracted from Rails).

4 Using Date and Time Form Helpers

You can choose not to use the form helpers generating HTML5 date and time input fields and use the alternative date and time helpers. These date and time helpers differ from all the other form helpers in two important respects:

- Dates and times are not representable by a single input element. Instead you have several, one for each component (year, month, day etc.) and so there is no single value in your `params` hash with your date or time.
- Other helpers use the `_tag` suffix to indicate whether a helper is a barebones helper or one that operates on model objects. With dates and times, `select_date`, `select_time` and `select_datetime` are the barebones helpers, `date_select`, `time_select` and `datetime_select` are the equivalent model object helpers.

Both of these families of helpers will create a series of select boxes for the different components (year, month, day etc.).

4.1 Barebones Helpers

The `select_*` family of helpers take as their first argument an instance of `Date`, `Time` or `DateTime` that is used as the currently selected value. You may omit this parameter, in which case the current date is used. For example:

```
<%= select_date Date.today, prefix: :start_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="start_date_year"
name="start_date[year]"> ... </select>
<select id="start_date_month"
name="start_date[month]"> ... </select>
<select id="start_date_day" name="start_date[day]"> ...
</select>
```

The above inputs would result in `params[:start_date]` being a hash with keys `:year`, `:month`, `:day`. To get an actual `Date`, `Time` or `DateTime` object you would have to extract these values and pass them to the appropriate constructor, for example:

```
Date.civil(params[:start_date][:year].to_i,
params[:start_date][:month].to_i, params[:start_date]
[:day].to_i)
```

The `:prefix` option is the key used to retrieve the hash of date components from the `params` hash. Here it was set to `start_date`, if omitted it will default to `date`.

4.2 Model Object Helpers

`select_date` does not work well with forms that update or create Active Record objects as Active Record expects each element of the `params` hash to correspond to one attribute. The model object helpers for dates and times submit parameters with special names; when Active Record sees parameters with such names it knows they must be combined with the other parameters and given to a constructor appropriate to the column type. For example:

```
<%= date_select :person, :birth_date %>
```

outputs (with actual option values omitted for brevity)

```
<select id="person_birth_date_1i"
name="person[birth_date(1i)]"> ... </select>
<select id="person_birth_date_2i"
name="person[birth_date(2i)]"> ... </select>
<select id="person_birth_date_3i"
name="person[birth_date(3i)]"> ... </select>
```

which results in a `params` hash like

```
{'person' => {'birth_date(1i)' => '2008',
'birth_date(2i)' => '11', 'birth_date(3i)' => '22'}}
```

When this is passed to `Person.new` (or `update`), Active Record spots that these parameters should all be used to construct the `birth_date` attribute and uses the suffixed information to determine in which order it should pass these parameters to functions such as `Date.civil`.

4.3 Common Options

Both families of helpers use the same core set of functions to generate the individual select tags and so both accept largely the same options. In particular, by default Rails will generate year options 5 years either side of the current year. If this is not an appropriate range, the `:start_year` and `:end_year` options override this. For an exhaustive list of the available options, refer to the API documentation.

As a rule of thumb you should be using `date_select` when working with model objects and `select_date` in other cases, such as a search form which filters results by date.

In many cases the built-in date pickers are clumsy as they do not aid the user in working out the relationship between the date and the day of the week.

4.4 Individual Components

Occasionally you need to display just a single date component such as a year or a month. Rails provides a series of helpers for this, one for each component `select_year`, `select_month`, `select_day`, `select_hour`, `select_minute`, `select_second`. These helpers are fairly straightforward. By default they will generate an input field named after the time component (for example, "year" for `select_year`, "month" for `select_month` etc.) although this can be overridden with the `:field_name` option. The `:prefix` option works in the same way that it does for `select_date` and `select_time` and has the same default value.

The first parameter specifies which value should be selected and can either be an instance of a `Date`, `Time` or `DateTime`, in which case the relevant component will be extracted, or a numerical value. For example:

```
<%= select_year(2009) %>
<%= select_year(Time.now) %>
```

will produce the same output if the current year is 2009 and the value chosen by the user can be retrieved by `params[:date][:year]`.

5 Uploading Files

A common task is uploading some sort of file, whether it's a picture of a person or a CSV file containing data to process. The most important thing to remember with file uploads is that the rendered form's encoding **MUST** be set to "multipart/form-data". If you use `form_for`, this is done automatically. If you use `form_tag`, you must set it yourself, as per the following example.

The following two forms both upload a file.

```
<%= form_tag({action: :upload}, multipart: true) do %>
  <%= file_field_tag 'picture' %>
<% end %>

<%= form_for @person do |f| %>
  <%= f.file_field :picture %>
<% end %>
```

Rails provides the usual pair of helpers: the barebones `file_field_tag` and the model oriented `file_field`. The only difference with other helpers is that you cannot set a default value for file inputs as this would have no meaning. As you would expect in the first case the uploaded file is in `params[:picture]` and in the second case in `params[:person][:picture]`.

5.1 What Gets Uploaded

The object in the `params` hash is an instance of a subclass of `IO`. Depending on the size of the uploaded file it may in fact be a `StringIO` or an instance of `File` backed by a temporary file. In both cases the object will have an original_filename attribute containing the name the file had on the user's computer and a content_type attribute containing the MIME type of the uploaded file. The following snippet saves the uploaded content in #{Rails.root}/public/uploads under the same name as the original file (assuming the form was the one in the previous example).

```
def upload
  uploaded_io = params[:person][:picture]
  File.open(Rails.root.join('public', 'uploads',
uploaded_io.original_filename), 'wb') do |file|
    file.write(uploaded_io.read)
  end
end
```

Once a file has been uploaded, there are a multitude of potential tasks, ranging from where to store the files (on disk, Amazon S3, etc) and associating them with models to resizing image files and generating thumbnails. The intricacies of this are beyond the scope of this guide, but there are several libraries designed to assist with these. Two of the better known ones are CarrierWave and Paperclip.

If the user has not selected a file the corresponding parameter will be an empty string.

5.2 Dealing with Ajax

Unlike other forms, making an asynchronous file upload form is not as simple as providing form_for with remote: true. With an Ajax form the serialization is done by JavaScript running inside the browser and since JavaScript cannot read files from your hard drive the file cannot be uploaded. The most common workaround is to use an invisible iframe that serves as the target for the form submission.

6 Customizing Form Builders

As mentioned previously the object yielded by form_for and fields_for is an instance of FormBuilder (or a subclass thereof). Form builders encapsulate the notion of displaying form elements for a single object. While you can of course write helpers for your forms in the usual way, you can also subclass FormBuilder and add the helpers there. For example:

```
<%= form_for @person do |f| %>
  <%= text_field_with_label f, :first_name %>
<% end %>
```

can be replaced with

```
<%= form_for @person, builder: LabellingFormBuilder do |
f| %>
  <%= f.text_field :first_name %>
<% end %>
```

by defining a LabellingFormBuilder class similar to the following:

```
class LabellingFormBuilder <
ActionView::Helpers::FormBuilder
  def text_field(attribute, options={})
    label(attribute) + super
  end
end
```

If you reuse this frequently you could define a `labeled_form_for` helper that automatically applies the `builder: LabellingFormBuilder` option:

```
def labeled_form_for(record, options = {}, &block)
  options.merge! builder: LabellingFormBuilder
  form_for record, options, &block
end
```

The form builder used also determines what happens when you do

```
<%= render partial: f %>
```

If `f` is an instance of `FormBuilder` then this will render the `form` partial, setting the partial's object to the form builder. If the form builder is of class `LabellingFormBuilder` then the `labelling_form` partial would be rendered instead.

7 Understanding Parameter Naming Conventions

As you've seen in the previous sections, values from forms can be at the top level of the `params` hash or nested in another hash. For example, in a standard `create` action for a Person model, `params[:person]` would usually be a hash of all the attributes for the person to create. The `params` hash can also contain arrays, arrays of hashes and so on.

Fundamentally HTML forms don't know about any sort of structured data, all they generate is name-value pairs, where pairs are just plain strings. The arrays and hashes you see in your application are the result of some parameter naming conventions that Rails uses.

7.1 Basic Structures

The two basic structures are arrays and hashes. Hashes mirror the syntax used for accessing the value in `params`. For example, if a form contains:

```
<input id="person_name" name="person[name]" type="text"
value="Henry"/>
```

the `params` hash will contain

```
{'person' => {'name' => 'Henry'}}
```

and `params[:person][:name]` will retrieve the submitted value in the controller.

Hashes can be nested as many levels as required, for example:

```
<input id="person_address_city" name="person[address]
[city]" type="text" value="New York"/>
```
will result in the `params` hash being
```
{'person' => {'address' => {'city' => 'New York'}}}
```
Normally Rails ignores duplicate parameter names. If the parameter name contains an empty set of square brackets `[]` then they will be accumulated in an array. If you wanted users to be able to input multiple phone numbers, you could place this in the form:
```
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
<input name="person[phone_number][]" type="text"/>
```
This would result in `params[:person][:phone_number]` being an array containing the inputted phone numbers.

7.2 Combining Them

We can mix and match these two concepts. One element of a hash might be an array as in the previous example, or you can have an array of hashes. For example, a form might let you create any number of addresses by repeating the following form fragment
```
<input name="addresses[][line1]" type="text"/>
<input name="addresses[][line2]" type="text"/>
<input name="addresses[][city]" type="text"/>
```
This would result in `params[:addresses]` being an array of hashes with keys `line1`, `line2` and `city`. Rails decides to start accumulating values in a new hash whenever it encounters an input name that already exists in the current hash.

There's a restriction, however, while hashes can be nested arbitrarily, only one level of "arrayness" is allowed. Arrays can usually be replaced by hashes; for example, instead of having an array of model objects, one can have a hash of model objects keyed by their id, an array index or some other parameter.

Array parameters do not play well with the `check_box` helper. According to the HTML specification unchecked checkboxes submit no value. However it is often convenient for a checkbox to always submit a value. The `check_box` helper fakes this by creating an auxiliary hidden input with the same name. If the checkbox is unchecked only the hidden input is submitted and if it is checked then both are submitted but the value submitted by the checkbox takes precedence. When working with array parameters this duplicate submission will confuse Rails since duplicate input names are how it decides when to start a new array element. It is preferable to either use `check_box_tag` or to use hashes instead of arrays.

7.3 Using Form Helpers

The previous sections did not use the Rails form helpers at all. While you can craft the input names yourself and pass them directly to helpers such as `text_field_tag` Rails also provides higher level support. The two tools at your disposal here are the name parameter to `form_for` and `fields_for` and the `:index` option that helpers take.
You might want to render a form with a set of edit fields for each of a person's addresses. For example:

```
<%= form_for @person do |person_form| %>
  <%= person_form.text_field :name %>
  <% @person.addresses.each do |address| %>
    <%= person_form.fields_for address, index:
address.id do |address_form|%>
      <%= address_form.text_field :city %>
    <% end %>
  <% end %>
<% end %>
```

Assuming the person had two addresses, with ids 23 and 45 this would create output similar to this:

```
<form accept-charset="UTF-8" action="/people/1"
class="edit_person" id="edit_person_1" method="post">
  <input id="person_name" name="person[name]"
type="text" />
  <input id="person_address_23_city"
name="person[address][23][city]" type="text" />
  <input id="person_address_45_city"
name="person[address][45][city]" type="text" />
</form>
```

This will result in a `params` hash that looks like

```
{'person' => {'name' => 'Bob', 'address' => {'23' =>
{'city' => 'Paris'}, '45' => {'city' => 'London'}}}}
```

Rails knows that all these inputs should be part of the person hash because you called `fields_for` on the first form builder. By specifying an `:index` option you're telling Rails that instead of naming the inputs `person[address][city]` it should insert that index surrounded by [] between the address and the city. This is often useful as it is then easy to locate which Address record should be modified. You can pass numbers with some other significance, strings or even `nil` (which will result in an array parameter being created).
To create more intricate nestings, you can specify the first part of the input name (`person[address]` in the previous example) explicitly:

```
<%= fields_for 'person[address][primary]', address,
index: address do |address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```
will create inputs like
```
<input id="person_address_primary_1_city"
name="person[address][primary][1][city]" type="text"
value="bologna" />
```
As a general rule the final input name is the concatenation of the name given to `fields_for`/`form_for`, the index value and the name of the attribute. You can also pass an `:index` option directly to helpers such as `text_field`, but it is usually less repetitive to specify this at the form builder level rather than on individual input controls.
As a shortcut you can append [] to the name and omit the `:index` option. This is the same as specifying `index: address` so
```
<%= fields_for 'person[address][primary][]', address do
|address_form| %>
  <%= address_form.text_field :city %>
<% end %>
```
produces exactly the same output as the previous example.

8 Forms to External Resources

Rails' form helpers can also be used to build a form for posting data to an external resource. However, at times it can be necessary to set an `authenticity_token` for the resource; this can be done by passing an `authenticity_token: 'your_external_token'` parameter to the `form_tag` options:
```
<%= form_tag 'http://farfar.away/form',
authenticity_token: 'external_token' do %>
  Form contents
<% end %>
```
Sometimes when submitting data to an external resource, like a payment gateway, the fields that can be used in the form are limited by an external API and it may be undesirable to generate an `authenticity_token`. To not send a token, simply pass `false` to the `:authenticity_token` option:
```
<%= form_tag 'http://farfar.away/form',
authenticity_token: false do %>
  Form contents
<% end %>
```
The same technique is also available for `form_for`:

```
<%= form_for @invoice, url: external_url,
authenticity_token: 'external_token' do |f| %>
  Form contents
<% end %>
```

Or if you don't want to render an `authenticity_token` field:

```
<%= form_for @invoice, url: external_url,
authenticity_token: false do |f| %>
  Form contents
<% end %>
```

9 Building Complex Forms

Many apps grow beyond simple forms editing a single object. For example, when creating a `Person` you might want to allow the user to (on the same form) create multiple address records (home, work, etc.). When later editing that person the user should be able to add, remove or amend addresses as necessary.

9.1 Configuring the Model

Active Record provides model level support via the `accepts_nested_attributes_for` method:

```
class Person < ApplicationRecord
  has_many :addresses, inverse_of: :person
  accepts_nested_attributes_for :addresses
end

class Address < ApplicationRecord
  belongs_to :person
end
```

This creates an `addresses_attributes=` method on `Person` that allows you to create, update and (optionally) destroy addresses.

9.2 Nested Forms

The following form allows a user to create a `Person` and its associated addresses.

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>

        <%= addresses_form.label :street %>
        <%= addresses_form.text_field :street %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

When an association accepts nested attributes `fields_for` renders its block once for every element of the association. In particular, if a person has no addresses it renders nothing. A common pattern is for the controller to build one or more empty children so that at least one set of fields is shown to the user. The example below would result in 2 sets of address fields being rendered on the new person form.

```
def new
  @person = Person.new
  2.times { @person.addresses.build }
end
```

The `fields_for` yields a form builder. The parameters' name will be what `accepts_nested_attributes_for` expects. For example, when creating a user with 2 addresses, the submitted parameters would look like:

```
{
  'person' => {
    'name' => 'John Doe',
    'addresses_attributes' => {
      '0' => {
        'kind' => 'Home',
        'street' => '221b Baker Street'
      },
      '1' => {
        'kind' => 'Office',
        'street' => '31 Spooner Street'
      }
    }
  }
}
```

The keys of the `:addresses_attributes` hash are unimportant, they need merely be different for each address.

If the associated object is already saved, `fields_for` autogenerates a hidden input with the `id` of the saved record. You can disable this by passing `include_id: false` to `fields_for`. You may wish to do this if the autogenerated input is placed in a location where an input tag is not valid HTML or when using an ORM where children do not have an `id`.

9.3 The Controller

As usual you need to whitelist the parameters in the controller before you pass them to the model:

```
def create
  @person = Person.new(person_params)
  # ...
end

private
  def person_params
    params.require(:person).permit(:name, 
addresses_attributes: [:id, :kind, :street])
  end
```

9.4 Removing Objects

You can allow users to delete associated objects by passing `allow_destroy: true` to `accepts_nested_attributes_for`

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, 
allow_destroy: true
end
```

If the hash of attributes for an object contains the key `_destroy` with a value of `1` or `true` then the object will be destroyed. This form allows users to remove addresses:

```
<%= form_for @person do |f| %>
  Addresses:
  <ul>
    <%= f.fields_for :addresses do |addresses_form| %>
      <li>
        <%= addresses_form.check_box :_destroy%>
        <%= addresses_form.label :kind %>
        <%= addresses_form.text_field :kind %>
        ...
      </li>
    <% end %>
  </ul>
<% end %>
```

Don't forget to update the whitelisted params in your controller to also include the `_destroy` field:

```
def person_params
  params.require(:person).
    permit(:name, addresses_attributes: 
[:id, :kind, :street, :_destroy])
end
```

9.5 Preventing Empty Records

It is often useful to ignore sets of fields that the user has not filled in. You can control this by passing a `:reject_if` proc to `accepts_nested_attributes_for`. This proc will be called with each hash of attributes submitted by the form. If the proc returns `false` then Active Record will not build an associated object for that hash. The example below only tries to build an address if the `kind` attribute is set.

```
class Person < ApplicationRecord
  has_many :addresses
  accepts_nested_attributes_for :addresses, reject_if: 
lambda {|attributes| attributes['kind'].blank?}
end
```

As a convenience you can instead pass the symbol `:all_blank` which will create a proc that will reject records where all the attributes are blank excluding any value for `_destroy`.

9.6 Adding Fields on the Fly

Rather than rendering multiple sets of fields ahead of time you may wish to add them only when a user clicks on an 'Add new address' button. Rails does not provide any built-in support for this. When generating new sets of fields you must ensure the key of the associated array is unique - the current JavaScript date (milliseconds after the epoch) is a common choice.

Rails Image Upload: Using CarrierWave in a Rails App

If you are building a web application, you definitely will want to enable image uploading. Image uploading is an important feature in modern-day applications, and images have been known to be useful in search engine optimization.
In this tutorial (which is the first part of the Rails Image Uploading series), I will show you how to enable image uploading in your Rails application using CarrierWave. It will be a simple application as the focus is on the image uploading.
CarrierWave is a Ruby gem that provides a simple and extremely flexible way to upload files from Ruby applications. You need to have Rails on your machine to follow along. To be sure, open up your terminal and enter the command below:

```
rails -v
```

That will show you the version of Rails you have installed. For this tutorial I will be using version 4.2.4, which you can install like so:

```
gem install rails -v 4.2.4
```

With that done, you are good to go.

Rails Application Setup

Now create a new Rails project:

```
rails new mypets
```

Open up your Gemfile and add the following gems.

```
*Gemfile*

...
gem 'carrierwave', '~> 0.10.0'
gem 'mini_magick', '~> 4.3'
...
```

The first gem is for CarrierWave, and the second gem called mini_magick helps with the resizing of images in your Rails application. With that done, run bundle install.

Generate a scaffold resource to add CarrierWave's functionality. Run the following command from your terminal:

```
rails g scaffold Pet name:string description:text image:string
```

A scaffold in Rails is a full set of model, database migration for that model, controller to manipulate it, views to view and manipulate the data, and a test suite for each of the above.

Migrate your database next:

```
rake db:migrate
```

Setting Up CarrierWave

You need to create an initializer for CarrierWave, which will be used for loading CarrierWave after loading ActiveRecord.

Navigate to **config > initializers** and create a file: carrier_wave.rb.

Paste the code below into it.
```
*config/initializers/carrier_wave.rb*

require 'carrierwave/orm/activerecord'
```
From your terminal, generate an uploader:
```
rails generate uploader Image
```
This will create a new directory called uploaders in the app folder and a file inside called `image_uploader.rb`. The content of the file should look like this:

01

Ruby on RailsRubyWeb Apps

```
rails -v
```
```
gem install rails -v 4.2.4
```
```
rails new mypets
```
```
*Gemfile*

...
gem 'carrierwave', '~> 0.10.0'
gem 'mini_magick', '~> 4.3'
...
```
```
rails g scaffold Pet name:string description:text
image:string
```
```
rake db:migrate
```

```
*config/initializers/carrier_wave.rb*

require 'carrierwave/orm/activerecord'
```

```
rails generate uploader Image
```

```
*app/uploaders/image_uploader.rb*

# encoding: utf-8

class ImageUploader < CarrierWave::Uploader::Base

  # Include RMagick or MiniMagick support:
  # include CarrierWave::RMagick
  # include CarrierWave::MiniMagick

  # Choose what kind of storage to use for this uploader:
  storage :file
  # storage :fog

   # Override the directory where uploaded files will be
stored.
   # This is a sensible default for uploaders that are
meant to be mounted:
  def store_dir
                "uploads/#{model.class.to_s.underscore}/
#{mounted_as}/#{model.id}"
  end

   # Provide a default URL as a default if there hasn't
been a file uploaded:
  # def default_url
  #   # For Rails 3.1+ asset pipeline compatibility:
```

```
                    #                                       #
ActionController::Base.helpers.asset_path("fallback/"   +
[version_name, "default.png"].compact.join('_'))
  #
     #          "/images/fallback/"   +   [version_name,
"default.png"].compact.join('_')
  # end

  # Process files as they are uploaded:
  # process :scale => [200, 300]

 #
  # def scale(width, height)
  #   # do something
  # end

  # Create different versions of your uploaded files:
  # version :thumb do
  #   process :resize_to_fit => [50, 50]
  # end

  # Add a white list of extensions which are allowed to
be uploaded.
  # For images you might use something like this:
  # def extension_white_list
  #   %w(jpg jpeg gif png)
  # end

  # Override the filename of the uploaded files:
   #  Avoid  using  model.id  or  version_name  here,  see
uploader/store.rb for details.
  # def filename
  #   "something.jpg" if original_filename
  # end

end
```

```
include CarrierWave::MiniMagick
```

```
version :thumb do
  process :resize_to_fill => [50, 50]
end
```

```
def extension_white_list
  %w(jpg jpeg gif png)
end
```

```
*app/model/pet.rb*
```

```
mount_uploader :image, ImageUploader
```

```
app/views/pets/_form.html.erb
```

```
<%= form_for @pet, html: { multipart: true } do |f| %>
  <% if @pet.errors.any? %>
    <div id="error_explanation">
        <h2><%= pluralize(@pet.errors.count, "error") %>
prohibited this pet from being saved:</h2>

      <ul>
      <% @pet.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :image %><br>
    <%= f.file_field :image %>
  </div>
  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

```
*app/views/pets/show.html.erb*

<p id="notice"><%= notice %></p>

<p>
  <strong>Name:</strong>
  <%= @pet.name %>
</p>

<p>
  <strong>Description:</strong>
  <%= @pet.description %>
</p>

<p>
  <strong>Image:</strong>
  <%= image_tag @pet.image.thumb.url %>
</p>

<%= link_to 'Edit', edit_pet_path(@pet) %> |
<%= link_to 'Back', pets_path %>
```

```
*app/views/pets/_form.html.erb*
<%= form_for @pet, html: { multipart: true } do |f| %>
  <% if @pet.errors.any? %>
    <div id="error_explanation">
        <h2><%= pluralize(@pet.errors.count, "error") %>
prohibited this pet from being saved:</h2>

      <ul>
      <% @pet.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= f.label :name %><br>
    <%= f.text_field :name %>
  </div>
  <div class="field">
    <%= f.label :description %><br>
    <%= f.text_area :description %>
  </div>
  <div class="field">
    <%= f.label :image %><br>
    <%= f.file_field :image %>
    <% if f.object.image? %>
      <%= image_tag f.object.image.thumb.url %>
      <%= f.label :remove_image %>
      <%= f.check_box :remove_image %>
    <% end %>
  </div>

  <div class="actions">
    <%= f.submit %>
  </div>
<% end %>
```

```
*app/model/pet.rb
validates_processing_of :image
validate :image_size_validation

private
  def image_size_validation
      errors[:image] << "should be less than 500KB" if
image.size > 0.5.megabytes
  end
```

Layouts and Rendering in Rails

1 Overview: How the Pieces Fit Together

How does the interaction between Controller and View in the Model-View-Controller triangle takes place?.

As you know, the Controller is responsible for orchestrating the whole process of handling a request in Rails, though it normally hands off any heavy code to the Model. But then, when it's time to send a response back to the user, the Controller hands things off to the View.

In broad strokes, this involves deciding what should be sent as the response and calling an appropriate method to create that response. If the response is a full-blown view, Rails also does some extra work to wrap the view in a layout and possibly to pull in partial views. You'll see all of those paths later in this guide.

2 Creating Responses

From the controller's point of view, there are three ways to create an HTTP response:

- Call **`render`** to create a full response to send back to the browser
- Call **`redirect_to`** to send an HTTP redirect status code to the browser
- Call **`head`** to create a response consisting solely of HTTP headers to send back to the browser

2.1 Rendering by Default: Convention Over Configuration in Action

By default, controllers in Rails automatically render views with names that correspond to valid routes.

For example, if you have this code in your `BooksController` class:

```
class BooksController < ApplicationController
end
```

And the following in your routes file:

```
resources :books
```

And you have a view file `app/views/books/index.html.erb`:

```
<h1>Books are coming soon!</h1>
```

Rails will automatically render `app/views/books/index.html.erb` when you navigate to `/books` and you will see "Books are coming soon!" on your screen.

However a coming soon screen is only minimally useful, so you will soon create your `Book` model and add the index action to `BooksController`:

```
class BooksController < ApplicationController
  def index
    @books = Book.all
  end
end
```

Note that we don't have explicit render at the end of the index action in accordance with "convention over configuration" principle. The rule is that if you do not explicitly render something at the end of a controller action, **Rails will automatically look for the `action_name.html.erb` template in the controller's view path and render it. So in this case, Rails will render the `app/views/books/index.html.erb` file.**

If we want to display the properties of all the books in our view, we can do so with an ERB template like this:

```
<h1>Listing Books</h1>

<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Content</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @books.each do |book| %>
      <tr>
        <td><%= book.title %></td>
        <td><%= book.content %></td>
        <td><%= link_to "Show", book %></td>
        <td><%= link_to "Edit", edit_book_path(book) 
%></td>
        <td><%= link_to "Destroy", book, 
method: :delete, data: { confirm: "Are you sure?" } %></
td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to "New book", new_book_path %>
```

2.2 Using render

In most cases, the `ActionController::Base#render` method does the heavy lifting of rendering your application's content for use by a browser. There are a variety of ways to customize the behavior of `render`. You can render the default view for a Rails template, or a specific template, or a file, or inline code, or nothing at all. You can render text, JSON, or XML. You can specify the content type or HTTP status of the rendered response as well.

If you want to see the exact results of a call to `render` without needing to inspect it in a browser, you can call `render_to_string`. This method takes exactly the same options as `render`, but it returns a string instead of sending a response back to the browser.

2.2.1 Rendering an Action's View

If you want to render the view that corresponds to a different template within the same controller, you can use `render` with the name of the view:

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render "edit"
  end
end
```

If the call to `update` fails, calling the `update` action in this controller will render the `edit.html.erb` template belonging to the same controller.

If you prefer, you can use a symbol instead of a string to specify the action to render:

```
def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
    redirect_to(@book)
  else
    render :edit
  end
end
```

2.2.2 Rendering an Action's Template from Another Controller

What if you want to render a template from an entirely different controller from the one that contains the action code? You can also do that with `render`, which accepts the full path (relative to `app/views`) of the template to render. For example, if you're running code in an `AdminProductsController` that lives in `app/controllers/admin`, you can render the results of an action to a template in `app/views/products` this way:

```
render "products/show"
```

Rails knows that this view belongs to a different controller because of the embedded slash character in the string. If you want to be explicit, you can use the `:template` option (which was required on Rails 2.2 and earlier):

```
render template: "products/show"
```

2.2.3 Rendering an Arbitrary File

The `render` method can also use a view that's entirely outside of your application:

```
render file: "/u/apps/warehouse_app/current/app/views/
products/show"
```

The `:file` option takes an absolute file-system path. Of course, you need to have rights to the view that you're using to render the content.

Using the `:file` option in combination with users input can lead to security problems since an attacker could use this action to access security sensitive files in your file system.

By default, the file is rendered using the current layout.

If you're running Rails on Microsoft Windows, you should use the `:file` option to render a file, because Windows filenames do not have the same format as Unix filenames.

2.2.4 Wrapping it up

The above three ways of rendering (rendering another template within the controller, rendering a template within another controller and rendering an arbitrary file on the file system) are actually variants of the same action.

In fact, in the BooksController class, inside of the update action where we want to render the edit template if the book does not update successfully, all of the following render calls would all render the `edit.html.erb` template in the `views/books` directory:

```
render :edit
render action: :edit
render "edit"
render "edit.html.erb"
render action: "edit"
render action: "edit.html.erb"
render "books/edit"
render "books/edit.html.erb"
render template: "books/edit"
render template: "books/edit.html.erb"
render "/path/to/rails/app/views/books/edit"
render "/path/to/rails/app/views/books/edit.html.erb"
render file: "/path/to/rails/app/views/books/edit"
render file: "/path/to/rails/app/views/books/
edit.html.erb"
```

Which one you use is really a matter of style and convention, but the rule of thumb is to use the simplest one that makes sense for the code you are writing.

2.2.5 Using `render` with `:inline`

The `render` method can do without a view completely, if you're willing to use the `:inline` option to supply ERB as part of the method call. This is perfectly valid:

```
render inline: "<% products.each do |p| %><p><%= p.name
%></p><% end %>"
```

There is seldom any good reason to use this option. Mixing ERB into your controllers defeats the MVC orientation of Rails and will make it harder for other developers to follow the logic of your project. Use a separate erb view instead.

By default, inline rendering uses ERB. You can force it to use Builder instead with the `:type` option:

```
render inline: "xml.p {'Horrid coding practice!'}",
type: :builder
```

2.2.6 Rendering Text

You can send plain text - with no markup at all - back to the browser by using the `:plain` option to `render`:

```
render plain: "OK"
```

Rendering pure text is most useful when you're responding to Ajax or web service requests that are expecting something other than proper HTML.

By default, if you use the `:plain` option, the text is rendered without using the current layout. If you want Rails to put the text into the current layout, you need to add the `layout: true` option and use the `.text.erb` extension for the layout file.

2.2.7 Rendering HTML

You can send an HTML string back to the browser by using the `:html` option to `render`:

```
render html: helpers.tag.strong('Not Found')
```

This is useful when you're rendering a small snippet of HTML code. However, you might want to consider moving it to a template file if the markup is complex.

When using `html:` option, HTML entities will be escaped if the string is not composed with `html_safe`-aware APIs.

2.2.8 Rendering JSON

JSON is a JavaScript data format used by many Ajax libraries. Rails has built-in support for converting objects to JSON and rendering that JSON back to the browser:

```
render json: @product
```

You don't need to call `to_json` on the object that you want to render. If you use the `:json`option, `render` will automatically call `to_json` for you.

2.2.9 Rendering XML

Rails also has built-in support for converting objects to XML and rendering that XML back to the caller:

```
render xml: @product
```

You don't need to call `to_xml` on the object that you want to render. If you use the `:xml`option, `render` will automatically call `to_xml` for you.

2.2.10 Rendering Vanilla JavaScript

Rails can render vanilla JavaScript:

```
render js: "alert('Hello Rails');"
```

This will send the supplied string to the browser with a MIME type of `text/javascript`.

2.2.11 Rendering raw body

You can send a raw content back to the browser, without setting any content type, by using the `:body` option to `render`:

```
render body: "raw"
```

This option should be used only if you don't care about the content type of the response. Using `:plain` or `:html` might be more appropriate most of the time.

Unless overridden, your response returned from this render option will be `text/plain`, as that is the default content type of Action Dispatch response.

2.2.12 Options for `render`

Calls to the `render` method generally accept five options:

- `:content_type`
- `:layout`

- `:location`
- `:status`
- `:formats`

2.2.12.1 The `:content_type` Option

By default, Rails will serve the results of a rendering operation with the MIME content-type of `text/html` (or `application/json` if you use the `:json` option, or `application/xml` for the `:xml` option.). There are times when you might like to change this, and you can do so by setting the `:content_type` option:

```
render file: filename, content_type: "application/rss"
```

2.2.12.2 The `:layout` Option

With most of the options to `render`, the rendered content is displayed as part of the current layout.

You can use the `:layout` option to tell Rails to use a specific file as the layout for the current action:

```
render layout: "special_layout"
```

You can also tell Rails to render with no layout at all:

```
render layout: false
```

2.2.12.3 The `:location` Option

You can use the `:location` option to set the HTTP `Location` header:

```
render xml: photo, location: photo_url(photo)
```

2.2.12.4 The `:status` Option

Rails will automatically generate a response with the correct HTTP status code (in most cases, this is `200 OK`). You can use the `:status` option to change this:

```
render status: 500
render status: :forbidden
```

Rails understands both numeric status codes and the corresponding symbols shown below.

Response Class	HTTP Status Code	Symbol
Informational	100	:continue
	101	:switching_protocols
	102	:processing
Success	200	:ok
	201	:created
	202	:accepted
	203	:non_authoritative_information
	204	:no_content
	205	:reset_content
	206	:partial_content
	207	:multi_status
	208	:already_reported
	226	:im_used
Redirection	300	:multiple_choices
	301	:moved_permanently
	302	:found
	303	:see_other

	304	:not_modified
	305	:use_proxy
	307	:temporary_redirect
	308	:permanent_redirect
Client Error	400	:bad_request
	401	:unauthorized
	402	:payment_required
	403	:forbidden
	404	:not_found
	405	:method_not_allowed
	406	:not_acceptable
	407	:proxy_authentication_required
	408	:request_timeout
	409	:conflict
	410	:gone
	411	:length_required
	412	:precondition_failed
	413	:payload_too_large
	414	:uri_too_long
	415	:unsupported_media_type

	416	:range_not_satisfiable
	417	:expectation_failed
	421	:misdirected_request
	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
	451	:unavailable_for_legal_reasons
Server Error	500	:internal_server_error
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported
	506	:variant_also_negotiates
	507	:insufficient_storage

	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

If you try to render content along with a non-content status code (100-199, 204, 205 or 304), it will be dropped from the response.

2.2.12.5 The `:formats` Option

Rails uses the format specified in the request (or `:html` by default). You can change this passing the `:formats` option with a symbol or an array:

```
render formats: :xml
render formats: [:json, :xml]
```

If a template with the specified format does not exist an `ActionView::MissingTemplate` error is raised.

2.2.13 Finding Layouts

To find the current layout, Rails first looks for a file in `app/views/layouts` with the same base name as the controller. For example, rendering actions from the `PhotosController` class will use `app/views/layouts/photos.html.erb` (or `app/views/layouts/photos.builder`).

If there is no such controller-specific layout, Rails will use

`app/views/layouts/application.html.erb` or

`app/views/layouts/application.builder`.

If there is no `.erb` layout, Rails will use a `.builder` layout if one exists. Rails also provides several ways to more precisely assign specific layouts to individual controllers and actions.

2.2.13.1 Specifying Layouts for Controllers

You can override the default layout conventions in your controllers by using the `layout` declaration. For example:

```
class ProductsController < ApplicationController
  layout "inventory"
  #...
end
```

With this declaration, all of the views rendered by the `ProductsController` will use `app/views/layouts/inventory.html.erb` as their layout.

To assign a specific layout for the entire application, use a `layout` declaration in your `ApplicationController` class:

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

With this declaration, all of the views in the entire application will use `app/views/layouts/main.html.erb` for their layout.

2.2.13.2 Choosing Layouts at Runtime

You can use a symbol to defer the choice of layout until a request is processed:

```
class ProductsController < ApplicationController
  layout :products_layout

  def show
    @product = Product.find(params[:id])
  end

  private
    def products_layout
      @current_user.special? ? "special" : "products"
    end

end
```

Now, if the current user is a special user, they'll get a special layout when viewing a product.

You can even use an inline method, such as a Proc, to determine the layout. For example, if you pass a Proc object, the block you give the Proc will be given the `controller` instance, so the layout can be determined based on the current request:

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr?
? "popup" : "application" }
end
```

2.2.13.3 Conditional Layouts

Layouts specified at the controller level support the `:only` and `:except` options. These options take either a method name, or an array of method names, corresponding to method names within the controller:

```
class ProductsController < ApplicationController
  layout "product", except: [:index, :rss]
end
```

With this declaration, the `product` layout would be used for everything but the `rss` and `index` methods.

2.2.13.4 Layout Inheritance

Layout declarations cascade downward in the hierarchy, and more specific layout declarations always override more general ones. For example:

- `application_controller.rb`

```ruby
class ApplicationController < ActionController::Base
  layout "main"
end
```

- `articles_controller.rb`

```ruby
class ArticlesController < ApplicationController
end
```

- `special_articles_controller.rb`

```ruby
class SpecialArticlesController < ArticlesController
  layout "special"
end
```

- `old_articles_controller.rb`

```ruby
class OldArticlesController < SpecialArticlesController
  layout false

  def show
    @article = Article.find(params[:id])
  end

  def index
    @old_articles = Article.older
    render layout: "old"
  end
  # ...
end
```

In this application:

- In general, views will be rendered in the `main` layout
- `ArticlesController#index` will use the `main` layout
- `SpecialArticlesController#index` will use the `special` layout
- `OldArticlesController#show` will use no layout at all
- `OldArticlesController#index` will use the `old` layout

2.2.13.5 Template Inheritance

Similar to the Layout Inheritance logic, if a template or partial is not found in the conventional path, the controller will look for a template or partial to render in its inheritance chain. For example:

```
# in app/controllers/application_controller
class ApplicationController < ActionController::Base
end

# in app/controllers/admin_controller
class AdminController < ApplicationController
end

# in app/controllers/admin/products_controller
class Admin::ProductsController < AdminController
  def index
  end
end
```

The lookup order for an `admin/products#index` action will be:

- `app/views/admin/products/`
- `app/views/admin/`
- `app/views/application/`

This makes `app/views/application/` a great place for your shared partials, which can then be rendered in your ERB as such:

```
<%# app/views/admin/products/index.html.erb %>
<%= render @products || "empty_list" %>

<%# app/views/application/_empty_list.html.erb %>
There are no items in this list <em>yet</em>.
```

2.2.14 Avoiding Double Render Errors

Sooner or later, most Rails developers will see the error message "Can only render or redirect once per action". While this is annoying, it's relatively easy to fix. Usually it happens because of a fundamental misunderstanding of the way that `render` works.

For example, here's some code that will trigger this error:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
  render action: "regular_show"
end
```

If `@book.special?` evaluates to `true`, Rails will start the rendering process to dump the `@book`variable into the `special_show` view. But this will *not* stop the rest of the code in the `show` action from running, and when Rails hits the end of the action, it will start to render the `regular_show` view - and throw an error. The solution is simple: make sure that you have only one call to `render` or `redirect` in a single code path. One thing that can help is `and return`. Here's a patched version of the method:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show" and return
  end
  render action: "regular_show"
end
```

Make sure to use `and return` instead of `&& return` because `&& return` will not work due to the operator precedence in the Ruby Language.

Note that the implicit render done by ActionController detects if `render` has been called, so the following will work without errors:

```
def show
  @book = Book.find(params[:id])
  if @book.special?
    render action: "special_show"
  end
end
```

This will render a book with `special?` set with the `special_show` template, while other books will render with the default `show` template.

2.3 Using `redirect_to`

Another way to handle returning responses to an HTTP request is with `redirect_to`. As you've seen, `render` tells Rails which view (or other asset) to use in constructing a response.

The `redirect_to` tells the browser to send a new request for a different URL.

For example, you could redirect from wherever you are in your code to the index of photos in your application with this call:

```
redirect_to photos_url
```

You can use `redirect_back` to return the user to the page they just came from. This location is pulled from the `HTTP_REFERER` header which is not guaranteed to be set by the browser, so you must provide the `fallback_location` to use in this case.

```
redirect_back(fallback_location: root_path)
```

`redirect_to` and `redirect_back` do not halt and return immediately from method execution, but simply set HTTP responses. Statements occurring after them in a method will be executed. You can halt by an explicit `return` or some other halting mechanism, if needed.

2.3.1 Getting a Different Redirect Status Code

Rails uses HTTP status code 302, a temporary redirect, when you call `redirect_to`. If you'd like to use a different status code, perhaps 301, a permanent redirect, you can use the `:status` option:

```
redirect_to photos_path, status: 301
```

Just like the `:status` option for `render`, `:status` for `redirect_to` accepts both numeric and symbolic header designations.

2.3.2 The Difference Between `render` and `redirect_to`

Sometimes inexperienced developers think of `redirect_to` as a sort of `goto` command, moving execution from one place to another in your Rails code. This is *not* correct. Your code stops running and waits for a new request for the browser. It just happens that you've told the browser what request it should make next, by sending back an HTTP 302 status code.

Consider these actions to see the difference:

```ruby
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    render action: "index"
  end
end
```

With the code in this form, there will likely be a problem if the `@book` variable is `nil`. Remember, a `render :action` doesn't run any code in the target action, so nothing will set up the `@books` variable that the `index` view will probably require. One way to fix this is to redirect instead of rendering:

```ruby
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    redirect_to action: :index
  end
end
```

With this code, the browser will make a fresh request for the index page, the code in the `index` method will run, and all will be well.

The only downside to this code is that it requires a round trip to the browser: the browser requested the show action with `/books/1` and the controller finds that there are no books, so the controller sends out a 302 redirect response to the browser telling it to go to `/books/`, the browser complies and sends a new request back to the controller asking now for the `index` action, the controller then gets all the books in the database and renders the index template, sending it back down to the browser which then shows it on your screen.

While in a small application, this added latency might not be a problem, it is something to think about if response time is a concern. We can demonstrate one way to handle this with a contrived example:

```ruby
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
    @books = Book.all
    flash.now[:alert] = "Your book was not found"
    render "index"
  end
end
```

This would detect that there are no books with the specified ID, populate the `@books` instance variable with all the books in the model, and then directly render the `index.html.erb` template, returning it to the browser with a flash alert message to tell the user what happened.

2.4 Using head To Build Header-Only Responses

The `head` method can be used to send responses with only headers to the browser. The `head` method accepts a number or symbol (see reference table) representing an HTTP status code. The options argument is interpreted as a hash of header names and values. For example, you can return only an error header:

```ruby
head :bad_request
```

This would produce the following header:

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Or you can use other HTTP headers to convey other information:

```
head :created, location: photo_path(@photo)
```

Which would produce:

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

3 Structuring Layouts

When Rails renders a view as a response, it does so by combining the view with the current layout, using the rules for finding the current layout that were covered earlier in this guide. Within a layout, you have access to three tools for combining different bits of output to form the overall response:

- Asset tags
- `yield` and `content_for`
- Partials

3.1 Asset Tag Helpers

Asset tag helpers provide methods for generating HTML that link views to feeds, JavaScript, stylesheets, images, videos, and audios. There are six asset tag helpers available in Rails:

- `auto_discovery_link_tag`
- `javascript_include_tag`
- `stylesheet_link_tag`
- `image_tag`
- `video_tag`
- `audio_tag`

You can use these tags in layouts or other views, although the `auto_discovery_link_tag`, `javascript_include_tag`, and `stylesheet_link_tag`, are most commonly used in the <head> section of a layout.

The asset tag helpers do *not* verify the existence of the assets at the specified locations; they simply assume that you know what you're doing and generate the link.

3.1.1 Linking to Feeds with the `auto_discovery_link_tag`

The `auto_discovery_link_tag` helper builds HTML that most browsers and feed readers can use to detect the presence of RSS, Atom, or JSON feeds. It takes the type of the link (`:rss`, `:atom`, or `:json`), a hash of options that are passed through to url_for, and a hash of options for the tag:

```
<%= auto_discovery_link_tag(:rss, {action: "feed"},
  {title: "RSS Feed"}) %>
```

Note: RSS **(originally RDF Site Summary; later, two competing approaches emerged, which used the backronyms Rich Site Summary and Really Simple Syndication respectively)** is a type of web feed which allows users and applications to access updates to online content in a standardized, computer-readable format.

There are three tag options available for the `auto_discovery_link_tag`:

- `:rel` specifies the `rel` value in the link. The default value is "alternate".
- `:type` specifies an explicit MIME type. Rails will generate an appropriate MIME type automatically.
- `:title` specifies the title of the link. The default value is the uppercase `:type` value, for example, "ATOM" or "RSS".

3.1.2 Linking to JavaScript Files with the `javascript_include_tag`

The `javascript_include_tag` helper returns an HTML `script` tag for each source provided.

If you are using Rails with the Asset Pipeline enabled, this helper will generate a link to `/assets/javascripts/` rather than `public/javascripts` which was used in earlier versions of Rails. This link is then served by the asset pipeline.

A JavaScript file within a Rails application or Rails engine goes in one of three locations: `app/assets`, `lib/assets` or `vendor/assets`. These locations are explained in detail in the Asset Organization section in the Asset Pipeline Guide.

You can specify a full path relative to the document root, or a URL, if you prefer. For example, to link to a JavaScript file that is inside a directory called `javascripts` inside of one of `app/assets`, `lib/assets` or `vendor/assets`, you would do this:

```
<%= javascript_include_tag "main" %>
```

Rails will then output a `script` tag such as this:

```
<script src='/assets/main.js'></script>
```

The request to this asset is then served by the Sprockets gem.

To include multiple files such as `app/assets/javascripts/main.js` and `app/assets/javascripts/columns.js` at the same time:

```
<%= javascript_include_tag "main", "columns" %>
```

To include `app/assets/javascripts/main.js` and `app/assets/javascripts/photos/columns.js`:

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

To include `http://example.com/main.js`:

```
<%= javascript_include_tag "http://example.com/main.js"
%>
```

3.1.3 Linking to CSS Files with the `stylesheet_link_tag`

The `stylesheet_link_tag` helper returns an HTML `<link>` tag for each source provided.

If you are using Rails with the "Asset Pipeline" enabled, this helper will generate a link to `/assets/stylesheets/`. This link is then processed by the Sprockets gem. A stylesheet file can be stored in one of three locations: `app/assets`, `lib/assets` or `vendor/assets`.

You can specify a full path relative to the document root, or a URL. For example, to link to a stylesheet file that is inside a directory called `stylesheets` inside of one of `app/assets`, `lib/assets` or `vendor/assets`, you would do this:

```
<%= stylesheet_link_tag "main" %>
```

To include `app/assets/stylesheets/main.css` and `app/assets/stylesheets/columns.css`:

```
<%= stylesheet_link_tag "main", "columns" %>
```

To include `app/assets/stylesheets/main.css` and `app/assets/stylesheets/photos/columns.css`:

```
<%= stylesheet_link_tag "main", "photos/columns" %>
```

To include `http://example.com/main.css`:

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

By default, the `stylesheet_link_tag` creates links with `media="screen" rel="stylesheet"`. You can override any of these defaults by specifying an appropriate option (`:media`, `:rel`):

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

3.1.4 Linking to Images with the `image_tag`

The `image_tag` helper builds an HTML `<img />` tag to the specified file. By default, files are loaded from `public/images`.

Note that you must specify the extension of the image.

```
<%= image_tag "header.png" %>
```

You can supply a path to the image if you like:

```
<%= image_tag "icons/delete.gif" %>
```

You can supply a hash of additional HTML options:

```
<%= image_tag "icons/delete.gif", {height: 45} %>
```

You can supply alternate text for the image which will be used if the user has images turned off in their browser. If you do not specify an alt text explicitly, it defaults to the file name of the file, capitalized and with no extension. For example, these two image tags would return the same code:

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", alt: "Home" %>
```

You can also specify a special size tag, in the format "{width}x{height}":

```
<%= image_tag "home.gif", size: "50x20" %>
```

In addition to the above special tags, you can supply a final hash of standard HTML options, such as `:class`, `:id` or `:name`:

```
<%= image_tag "home.gif", alt: "Go Home",
                          id: "HomeImage",
                          class: "nav_bar" %>
```

3.1.5 Linking to Videos with the `video_tag`

The `video_tag` helper builds an HTML 5 `<video>` tag to the specified file. By default, files are loaded from `public/videos`.

```
<%= video_tag "movie.ogg" %>
```

Produces

```
<video src="/videos/movie.ogg" />
```

Like an `image_tag` you can supply a path, either absolute, or relative to the `public/videos`directory. Additionally you can specify the `size: "#{width}x#{height}"` option just like an `image_tag`. Video tags can also have any of the HTML options specified at the end (`id`, `class` et al).

The video tag also supports all of the `<video>` HTML options through the HTML options hash, including:

- `poster: "image_name.png"`, provides an image to put in place of the video before it starts playing.
- `autoplay: true`, starts playing the video on page load.
- `loop: true`, loops the video once it gets to the end.
- `controls: true`, provides browser supplied controls for the user to interact with the video.
- `autobuffer: true`, the video will pre load the file for the user on page load.

You can also specify multiple videos to play by passing an array of videos to the `video_tag`:

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

This will produce:

```
<video>
  <source src="/videos/trailer.ogg">
  <source src="/videos/movie.ogg">
</video>
```

3.1.6 Linking to Audio Files with the audio_tag

The `audio_tag` helper builds an HTML 5 `<audio>` tag to the specified file. By default, files are loaded from `public/audios`.

```
<%= audio_tag "music.mp3" %>
```

You can supply a path to the audio file if you like:

```
<%= audio_tag "music/first_song.mp3" %>
```

You can also supply a hash of additional options, such as `:id`, `:class` etc.

Like the `video_tag`, the `audio_tag` has special options:

- `autoplay: true`, starts playing the audio on page load
- `controls: true`, provides browser supplied controls for the user to interact with the audio.
- `autobuffer: true`, the audio will pre load the file for the user on page load.

3.2 Understanding yield

Within the context of a layout, `yield` identifies a section where content from the view should be inserted. The simplest way to use this is to have a single `yield`, into which the entire contents of the view currently being rendered is inserted:

```
<html>
  <head>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

You can also create a layout with multiple yielding regions:

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

The main body of the view will always render into the unnamed `yield`. To render content into a named `yield`, you use the `content_for` method.

3.3 Using the `content_for` Method

The `content_for` method allows you to insert content into a named `yield` block in your layout. For example, this view would work with the layout that you just saw:

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

The result of rendering this page into the supplied layout would be this HTML:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

The `content_for` method is very helpful when your layout contains distinct regions such as sidebars and footers that should get their own blocks of content inserted. It's also useful for inserting tags that load page-specific JavaScript or css files into the header of an otherwise generic layout.

3.4 Using Partials

Partial templates - usually just called "partials" - are another device for breaking the rendering process into more manageable chunks. With a partial, you can move the code for rendering a particular piece of a response to its own file.

3.4.1 Naming Partials

To render a partial as part of a view, you use the `render` method within the view:

```
<%= render "menu" %>
```

This will render a file named `_menu.html.erb` at that point within the view being rendered. Note the leading underscore character: partials are named with a leading underscore to distinguish them from regular views, even though they are referred to without the underscore. This holds true even when you're pulling in a partial from another folder:

```
<%= render "shared/menu" %>
```

That code will pull in the partial from `app/views/shared/_menu.html.erb`.

3.4.2 Using Partials to Simplify Views

One way to use partials is to treat them as the equivalent of subroutines: as a way to move details out of a view so that you can grasp what's going on more easily. For example, you might have a view that looked like this:

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

<%= render "shared/footer" %>
```

Here, the `_ad_banner.html.erb` and `_footer.html.erb` partials could contain content that is shared by many pages in your application. You don't need to see the details of these sections when you're concentrating on a particular page.

As seen in the previous sections of this guide, `yield` is a very powerful tool for cleaning up your layouts. Keep in mind that it's pure Ruby, so you can use it almost everywhere. For example, we can use it to DRY up form layout definitions for several similar resources:

- `users/index.html.erb`

```
<%= render "shared/search_filters", search: @q do |f|
%>
  <p>
    Name contains: <%= f.text_field :name_contains %>
  </p>
<% end %>
```

- `roles/index.html.erb`

```
• <%= render "shared/search_filters", search: @q do |f|
%>
    <p>
      Title contains: <%= f.text_field :title_contains %>
    </p>
  <% end %>


•     •
```

- shared/_search_filters.html.erb

```
• <%= form_for(search) do |f| %>
    <h1>Search form:</h1>
    <fieldset>
      <%= yield f %>
    </fieldset>
    <p>
      <%= f.submit "Search" %>
    </p>
  <% end %>


•     •
```

For content that is shared among all pages in your application, you can use partials directly from layouts.

3.4.3 Partial Layouts

A partial can use its own layout file, just as a view can use a layout. For example, you might call a partial like this:

```
<%= render partial: "link_area", layout: "graybar" %>
```

This would look for a partial named `_link_area.html.erb` and render it using the layout `_graybar.html.erb`. Note that layouts for partials follow the same leading-underscore naming as regular partials, and are placed in the same folder with the partial that they belong to (not in the master `layouts` folder).

Also note that explicitly specifying `:partial` is required when passing additional options such as `:layout`.

3.4.4 Passing Local Variables

You can also pass local variables into partials, making them even more powerful and flexible. For example, you can use this technique to reduce duplication between new and edit pages, while still keeping a bit of distinct content:

- `new.html.erb`

```
<h1>New zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

- `edit.html.erb`

```
<h1>Editing zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

- `_form.html.erb`

```
<%= form_for(zone) do |f| %>
  <p>
    <b>Zone name</b><br>
    <%= f.text_field :name %>
  </p>
  <p>
    <%= f.submit %>
  </p>
<% end %>
```

Although the same partial will be rendered into both views, Action View's submit helper will return "Create Zone" for the new action and "Update Zone" for the edit action.

To pass a local variable to a partial in only specific cases use the `local_assigns`.

- `index.html.erb`

```
<%= render user.articles %>
```

- `show.html.erb`

```
<%= render article, full: true %>
```

- `_article.html.erb`

```
<h2><%= article.title %></h2>

<% if local_assigns[:full] %>
  <%= simple_format article.body %>
<% else %>
  <%= truncate article.body %>
<% end %>
```

This way it is possible to use the partial without the need to declare all local variables.

Every partial also has a local variable with the same name as the partial (minus the leading underscore). You can pass an object in to this local variable via the `:object` option:

```
<%= render partial: "customer", object: @new_customer %>
```

Within the `customer` partial, the `customer` variable will refer to `@new_customer` from the parent view.

If you have an instance of a model to render into a partial, you can use a shorthand syntax:

```
<%= render @customer %>
```

Assuming that the `@customer` instance variable contains an instance of the `Customer` model, this will use `_customer.html.erb` to render it and will pass the local variable `customer` into the partial which will refer to the `@customer` instance variable in the parent view.

3.4.5 Rendering Collections

Partials are very useful in rendering collections. When you pass a collection to a partial via the `:collection` option, the partial will be inserted once for each member in the collection:

- `index.html.erb`

```
<h1>Products</h1>
<%= render partial: "product", collection: @products %>
```

- `_product.html.erb`

```
<p>Product Name: <%= product.name %></p>
```

When a partial is called with a pluralized collection, then the individual instances of the partial have access to the member of the collection being rendered via a variable named after the partial. In this case, the partial is `_product`, and within the `_product` partial, you can refer to `product` to get the instance that is being rendered.

There is also a shorthand for this. Assuming `@products` is a collection of `product` instances, you can simply write this in the `index.html.erb` to produce the same result:

```
<h1>Products</h1>
<%= render @products %>
```

Rails determines the name of the partial to use by looking at the model name in the collection. In fact, you can even create a heterogeneous collection and render it this way, and Rails will choose the proper partial for each member of the collection:

- `index.html.erb`

```
<h1>Contacts</h1>
<%= render [customer1, employee1, customer2, employee2]
%>
```

- `customers/_customer.html.erb`

```
<p>Customer: <%= customer.name %></p>
```

- `employees/_employee.html.erb`

```
<p>Employee: <%= employee.name %></p>
```

In this case, Rails will use the customer or employee partials as appropriate for each member of the collection.

In the event that the collection is empty, `render` will return nil, so it should be fairly simple to provide alternative content.

```
<h1>Products</h1>
<%= render(@products) || "There are no products
available." %>
```

3.4.6 Local Variables

To use a custom local variable name within the partial, specify the `:as` option in the call to the partial:

```
<%= render partial: "product", collection: @products,
as: :item %>
```

With this change, you can access an instance of the `@products` collection as the `item` local variable within the partial.

You can also pass in arbitrary local variables to any partial you are rendering with the `locals: {}`option:

```
<%= render partial: "product", collection: @products,
           as: :item, locals: {title: "Products Page"}
%>
```

In this case, the partial will have access to a local variable `title` with the value "Products Page".

Rails also makes a counter variable available within a partial called by the collection, named after the title of the partial followed by `_counter`. For example, when rendering a collection `@products` the partial `_product.html.erb` can access the variable `product_counter` which indexes the number of times it has been rendered within the enclosing view.

You can also specify a second partial to be rendered between instances of the main partial by using the `:spacer_template` option:

3.4.7 Spacer Templates

```
<%= render partial: @products, spacer_template:
"product_ruler" %>
```

Rails will render the `_product_ruler` partial (with no data passed in to it) between each pair of `_product` partials.

3.4.8 Collection Partial Layouts

When rendering collections it is also possible to use the `:layout` option:

```
<%= render partial: "product", collection: @products,
layout: "special_layout" %>
```

The layout will be rendered together with the partial for each item in the collection. The current object and object_counter variables will be available in the layout as well, the same way they are within the partial.

3.5 Using Nested Layouts

You may find that your application requires a layout that differs slightly from your regular application layout to support one particular controller. Rather than repeating the main layout and editing it, you can accomplish this by using nested layouts (sometimes called sub-templates). Here's an example:

Suppose you have the following `ApplicationController` layout:

- `app/views/layouts/application.html.erb`

```
• <html>
  <head>
    <title><%= @page_title or "Page Title" %></title>
    <%= stylesheet_link_tag "layout" %>
    <style><%= yield :stylesheets %></style>
  </head>
  <body>
    <div id="top_menu">Top menu items here</div>
    <div id="menu">Menu items here</div>
    <div id="content"><%= content_for?(:content) ?
  yield(:content) : yield %></div>
  </body>
  </html>


•     •
```

On pages generated by `NewsController`, you want to hide the top menu and add a right menu:

- `app/views/layouts/news.html.erb`

```
<% content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow;
color: black}
<% end %>
<% content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ?
yield(:news_content) : yield %>
<% end %>
<%= render template: "layouts/application" %>
```

That's it. The News views will use the new layout, hiding the top menu and adding a new right menu inside the "content" div.

There are several ways of getting similar results with different sub-templating schemes using this technique. Note that there is no limit in nesting levels. One can use the `ActionView::render`method via `render template: 'layouts/news'` to base a new layout on the News layout. If you are sure you will not subtemplate the `News` layout, you can replace the `content_for?(:news_content) ? yield(:news_content) : yield` with simply `yield`.

What is JSON?

JSON (JavaScript Object Notation) is a lightweight data-interchange format. More important, JSON is a human readable serialization format, like the popular YAML format all Rubyist are probably familiar with.

Compared to other serialization alternatives such as XML, YAML or Binary-serialization, JSON offers the following advantages:

- it's a human readable format
- it's largely adopted and supported by the most part of programming languages
- it's a language-independent format
- can be compressed in one line to reduce stream size
- can represent the most part of standard objects
- seamlessly integrates with JavaScript which makes JSON the standard for streaming data over AJAX calls

All these features make JSON an excellent serialization format. Of course, there are also some drawbacks, but this is material for an other article.

Installation

The library can be installed via rubygems:

```
# gem install json
```

If you have to use the pure variant, you can use:

```
# gem install json_pure
```

The gem and the source archive can also be downloaded directly from rubyforge.org.

Usage

If you require JSON like this:

```
require 'json'
```

JSON first tries to load the extension variant. If this fails, the pure variant is loaded and used.

To determine, which variant is active you can use the follwing methods:

- Ext variant: `[ JSON.parser, JSON.generator ] # => [JSON::Ext::Parser, JSON::Ext::Generator]`
-
- Pure variant: `[ JSON.parser, JSON.generator ] # => [JSON::Pure::Parser, JSON::Pure::Generator]`
-

If you want to enforce loading of a special variant, use

```
require 'json/ext'
```

to load the extension variant. Or use

```
require 'json/pure'
```

to use the pure variant.

You can choose to load a set of common additions to ruby core's objects if you

```
  require 'json/add/core'
```

To get the best compatibility to rails' JSON implementation, you can

```
    require 'json/add/rails'
```
Both of the additions attempt to require 'json' (like above) first, if it has not been required yet.

```
JSON.parse(document)
```
If you want to generate a JSON document from a ruby data structure call

```
JSON.generate(data)
```
You can also use the `pretty_generate` method (which formats the output more verbosely and nicely) or `fast_generate`(which doesn't do any of the security checks generate performs, e. g. nesting deepness checks).

There are also the JSON and JSON[] methods which use parse on a String or generate a JSON document from an array or hash:

```
document = JSON 'test'  => 23 # => "{\"test\":23}"
document = JSON['test' => 23] # => "{\"test\":23}"
```
and

```
data = JSON '{"test":23}'  # => {"test"=>23}
data = JSON['{"test":23}'] # => {"test"=>23}
```

You can choose to load a set of common additions to ruby core's objects if you

```
require 'json/add/core'
```
After requiring this you can, e. g., serialise/deserialise Ruby ranges:

```
JSON JSON(1..10) # => 1..10
```
To find out how to add JSON support to other or your own classes, read the section "More Examples" below.

To get the best compatibility to rails' JSON implementation, you can

```
require 'json/add/rails'
```

Both of the additions attempt to require `'json'` (like above) first, if it has not been required yet.

Serializing exceptions

The JSON module doesn't extend `Exception` by default. If you convert an `Exception` object to JSON, it will by default only include the exception message.

To include the full details, you must either load the `json/add/core` mentioned above, or specifically load the exception addition:

```
require 'json/add/exception'
```

More Examples

To create a JSON document from a ruby data structure, you can call `JSON.generate` like that:

```
json = JSON.generate [1, 2, {"a"=>3.141}, false, true,
nil, 4..10]

# => “[1,2,{\"a\":3.141},false,true,null,\"4..10\"]"
```

To get back a ruby data structure from a JSON document, you have to call JSON.parse on it:

```
JSON.parse json
# => [1, 2, {"a"=>3.141}, false, true, nil, “4..10"]
```

Note, that the range from the original data structure is a simple string now. The reason for this is, that JSON doesn't support ranges or arbitrary classes. In this case the json library falls back to call `Object#to_json`, which is the same as `#to_s.to_json`.

It's possible to add JSON support serialization to arbitrary classes by simply implementing a more specialized version of the `#to_json method`, that should return a JSON object (a hash converted to

JSON with #to_json) like this (don't forget the *a for all the arguments):

```
class Range
  def to_json(*a)
    {
      'json_class'   => self.class.name, # = 'Range'
      'data'         => [ first, last, exclude_end? ]
    }.to_json(*a)
  end
end
```

The hash key json_class is the class, that will be asked to deserialise the JSON representation later. In this case it's Range, but any namespace of the form A::B or ::A::B will do. All other keys are arbitrary and can be used to store the necessary data to configure the object to be deserialised.

If the key json_class is found in a JSON object, the JSON parser checks if the given class responds to the json_createclass method. If so, it is called with the JSON object converted to a Ruby hash.

So a range can be deserialised by implementing Range.json_create like this:

```
class Range
  def self.json_create(o)
    new(*o['data'])
  end
end
```

Now it possible to serialise/deserialise ranges as well:

```
json =JSON.generate [1, 2, {"a"=>3.141}, false, true, nil,
4..10]
#  =>  "[1,2,{\"a\":3.141},false,true,null,{\"json_class\":
\"Range\",\"data\":[4,10,false]}]"

JSON.parse json
# => [1, 2, {"a"=>3.141}, false, true, nil, 4..10]
```

```
json = JSON.generate [1, 2, {"a"=>3.141}, false, true,
nil, 4..10]

# => "[1,2,{\"a\":3.141},false,true,null,{\"json_class\":
\"Range\",\"data\":[4,10,false]}]"

JSON.parse json, :create_additions => true
# => [1, 2, {"a"=>3.141}, false, true, nil, 4..10]
```

`JSON.generate` always creates the shortest possible string representation of a ruby data structure in one line. This is good for data storage or network protocols, but not so good for humans to read. Fortunately there's

also `JSON.pretty_generate`(or `JSON.pretty_generate`) that creates a more readable output:

```
 puts JSON.pretty_generate([1, 2, {"a"=>3.141}, false,
true, nil, 4..10])
 [
   1,
   2,
   {
     "a": 3.141
   },
   false,
   true,
   null,
   {
     "json_class": "Range",
     "data": [
       4,
       10,
       false
     ]
   }
 ]
```

There are also the methods `Kernel#j` for generate, and `Kernel#jj` for `pretty_generate` output to the console, that work analogous to Core Ruby's `p` and the `pp` library's `pp` methods.

JSON and Ruby on Rails

Ruby on Rails is a web application framework and JSON is strictly related to the web ecosystem as a subset of the JavaScript programming language. There are many different parts of a Ruby on Rails application where you might need to manipulate, encode and decode a JSON string into a Ruby object and vice-versa.

JSON support in Ruby on Rails is provided by the `ActiveSupport::JSON` module. Behind the scenes, `ActiveSupport` wraps the JSON library, a standard Ruby Gem which you can use in any Ruby project. However, `ActiveSupport` goes beyond the boundary of a simple wrapper: it provides a JSON definition for the most part of the Ruby objects making JSON an effective full drop-in replacement for YAML.

ActiveSupport::JSON

As I mentioned before, `ActiveSupport::JSON` relies on the JSON Gem thus you need to have both libraries installed on your system. If you installed the Ruby on Rails framework, then you already have everything you need to start working with JSON.

The module provides a super-simple API composed by two methods:

- `ActiveSupport::JSON.encode(object)`: takes a Ruby object as value and returns a JSON-encoded string.
- `ActiveSupport::JSON.decode(string)`: takes a JSON-encoded string and returns the corresponding Ruby object

Here's a few examples:

```
j = ActiveSupport::JSON
ruby-1.8.7-p249 > j.encode(23)
# => "23"
j.encode("A string")
# => "A string"
j.encode({ :color => ["red", "green", "jellow"]
})
# => {"color":["red","green","jellow"]}
j.encode({ :color => ["red", "green",
"jellow"], :date => Time.now })
#                => {"color":
["red","green","jellow"],"date":"2010-04-29T00:
28:56+02:00"}

j.decode(j.encode({ :color => ["red", "green",
"jellow"], :date => Time.now }))
# => {"date"=>"2010-04-29T00:25:52+02:00",
"color"=>["red", "green", "jellow"]}
```

As you can see, the usage is really straightforward and the JSON-encoded result size is smaller compared to the YAML and XML counterparts.

```
v = { :color => ["red", "green",
"jellow"], :date => Time.now }
```

```
# => {:color=>["red", "green", "jellow"], :date=>Thu Apr 29 00:28:56 +0200 2010}

ActiveSupport::JSON.encode(v)
# 69 bytes
# => {"color":["red","green","jellow"],"date":"2010-04-29T00:28:56+02:00"}

YAML.dump(v)
# ---
# :color:
# - red
# - green
# - jellow
# :date: 2016-08-06 13:08:09.592621000 +02:00
```

ActiveSupport::JSON vs JSON

At the beginning of the article, I said `ActiveSupport::JSON` is something more than a mere JSON wrapper. Now it's the time to explain that statement.

The JSON format natively supports only a limited subset of variable types such as `String`, `Number`, `Array` and `Hash`. Easy to understand, being a language-agnostic format, it doesn't support complex or ruby-specific objects such as `Object`, `Exception` or `Range`. For this reason, JSON library delegates to each class the implementation of the JSON representation of the object using the `to_json` method. Similar to other standard transformation methods such as `to_s` or `to_f`, `to_json` is supposed to return a JSON-compatible representation.

While the `JSON` library now ships with a number of prepackaged definitions, by default it doesn't support most of the standard Ruby objects. Also it doesn't support the serialization of ActiveRecord objects and, working with Rails projects and database records, this might be a huge limitation.

`ActiveSupport::JSON` solves this problem and provides a predefined `to_json` implementation for the most part of Ruby/Rails objects. It also defines a simple `Object#to_json` making virtually every Ruby object JSON-compatible.

As of ActiveSupport 2.3.5, the following classes are supported: `String`, `Symbol`, `Date`, `Time`, `DateTime`, `Enumerable`, `Array`, `Hash`, `FalseClass`, `TrueClass`, `NilClass`, `Numeric`, `Float`, `Integer`, `Regexp`, and `Object`. You can find them in the `lib/active_support/json/encoders` folder. The `ActiveRecord`serialization/deserialization strategy is defined in the ActiveRecord library in `lib/active_record/serializers/json_serializer.rb`:

```
def to_json(options = {})
  super
end

def as_json(options = nil) #:nodoc:
    hash = Serializer.new(self,
options).serializable_record
    hash = { self.class.model_name.element =>
hash } if include_root_in_json
  hash
end

def from_json(json)
  self.attributes =
ActiveSupport::JSON.decode(json)
  self
end
```

Compared with `JSON` Gem, `ActiveSupport::JSON` is the solution to the following Alan's statement:

> There is bad news of course, in that your objects won't automagically be converted to JSON, unless all you're using is hashes, arrays and primitives. You need to do a little bit of work to make sure your custom object is serializable. Let's make one of the classes we introduced previously serializable using JSON.

As a side note, it also provides some additional features such as an interchangeable encoding/decoding backend.

JSON with Ruby and Rails

JSON is a beautiful format for storing objects as human readable text. It’s succeeded where XML has failed. Not only is it not shit, it’s actually quite good! But don’t just take my word for it, have a look at some of the “cool” ways you can generate and consume JSON.

Ruby support for JSON

Ruby’s JSON library makes parsing and generating JSON simple.

Converting between hash and json in Ruby

```
$ irb

>> require 'json'

=> true

>> json_text = { :name => 'Mike', :age => 70 }.to_json

=> "{\"name\":\"Mike\",\"age\":70}"

>> JSON.parse(json_text)

=> {"name"=>"Mike", “age"=>70}
```

HTTParty helps with communicating with RESTful services

Here we grab a record from Facebook.

Retrieve a JSON Resource

```
$ irb

>> require 'awesome_print'

=> true

>> require 'json'

=> true

>> require 'httparty'

=> true

>> ap JSON.parse HTTParty.get('https://graph.facebook.com/Stoptheclock').response.body

{

                    "about" => "Abolish the 28 Day Rule for Victorian Shelters\n\nhttp://stoptheclock.com.au\n\ninfo@stoptheclock.com.au",

          "category" => "Community",

           "founded" => "2010",

        "is_published" => true,

           "mission" => "To bring an end to the law requiring Victorian shelters to kill healthy adoptable cats and dogs after four weeks.",
    "talking_about_count" => 3,
               "username" => "Stoptheclock",
                "website" => "http://stoptheclock.com.au",
        "were_here_count" => 0,
                     "id" => "167163086642552",
                   "name" => "Stop The Clock",
                   "link" => "http://www.facebook.com/Stoptheclock",
                  "likes" => 5517
}
=> nil
```

HTTParty gets Classy

Creating a simple class allows you to DRY things up a bit

```
$ irb
>> require 'httparty'
=> true
>> class Facebook
>>   include HTTParty
>>   base_uri 'https://graph.facebook.com/'
>>   # default_params :output => 'json'
?>   format :json
>>
?>   def self.object(id)
>>     get "/#{id}"
>>   end
>> end
=> nil
>>
>> require 'awesome_print'
>> ap Facebook.object('Stoptheclock').parsed_response
{
                          "about" => "Abolish the 28 Day Rule for Victorian Shelters\n\nhttp://stoptheclock.com.au\n\ninfo@stoptheclock.com.au",
             "category" => "Community",
              "founded" => "2010",
          "is_published" => true,
              "mission" => "To bring an end to the law requiring Victorian shelters to kill healthy adoptable cats and dogs after four weeks.",
    "talking_about_count" => 3,
             "username" => "Stoptheclock",
              "website" => "http://stoptheclock.com.au",
       "were_here_count" => 0,
                   "id" => "167163086642552",
                 "name" => "Stop The Clock",
                                 "link" => "http://www.facebook.com/Stoptheclock",
                "likes" => 5517
}
=> nil
```

Rails support for JSON

ActiveSupport::JSON knows how to convert ActiveRecord objects (and more) to JSON. Simone Carletti explains how this differs from the standard lib.

```
## Encode

json = ActiveSupport::JSON.encode(object) # extra methods like :include
json = Offering.first.to_json(:include => :outlet, :methods => [:days_waiting])
```

## Decode

```
ActiveSupport::JSON.decode(json)
```

Rails3 niceness

Adding JSON to your Rails3 app doesn’t require a lot of extra code. You can specify method calls and associated objects to include as well as restrict the attributes returned. Simple eh?

```
class PostController < ApplicationController

 respond_to :json, :html, :jpg, :xml


 def index

   respond_with(@posts = Post.all),

                :methods => [:average_rating],

                :include => :comments

 end


 def show

     respond_with(@post = Post.find(params[:id])), :only => [:name, :body]
```

```
  end

end
```

HTTP Requests in Ruby

If you'd like to get information from a website, or if you'd like to submit forms, upload files…

…you'll need to send an HTTP request & then process the response.

In this article you'll learn how to:

Make a simple HTTP request using net/http

Send SSL requests

Submit data using a POST request

Send custom headers

Choose the best HTTP client for your situation

Let's do this!

How to Send an HTTP Request

Ruby comes with a built-in http client, it's called net/http & you can use it to send any kind of request you need.

Here's a net/http example:

```
require 'net/http'
```

```
Net::HTTP.get('example.com', '/index.html')
```

This will return a string with the HTML content of the page.

But often you want more than the HTML content.

Like the HTTP response status.

Without the response status you don't know if your request was successful, or if it failed.

This is how you get that:

```
response = Net::HTTP.get_response('example.com', '/')

response.code

# 200
```

Now if you want the response content you call the body method:

```
response.body
```

How to Use the HTTParty Gem

There are many gems that can make things easier for you.

One of these gems is httparty.

Here's how to use it:

```
require 'httparty'
```

```
response = HTTParty.get('http://example.com')


response.code

# 200


response.body

# ...
```

The benefits of using an HTTP gem:

It’s easier to use.

There is no separate get_response method, get already gives you a response object.

As you’ll see in the next section they make SSL request transparent

Sending SSL Requests

If you try to send an SSL request with net/http:

```
Net::HTTP.get_response("example.com", "/", 443)
```

You get:

```
Errno::ECONNRESET: Connection reset by peer
```

You’d have to do this instead:

```
net = Net::HTTP.new("example.com", 443)
```

net.use_ssl = true

net.get_response("/")

Save yourself some work & use a gem

How to Submit Data With a Post Request

A GET request is used to request information.

Like downloading an image, css, javascript…

But if you want to submit information use a POST request.

Here's an example:

HTTParty.post("http://example.com/login", body: { user: "test@example.com", password: "chunky_bacon" })

To upload a file you'll need a multipart request, which is not supported by HTTParty.

You can use the rest client gem:

require 'rest-client'

RestClient.post '/profile', file: File.new('photo.jpg', 'rb')

Or the Faraday gem:

require 'faraday'

```
conn =
Faraday.new do |f|
  f.request :multipart
  f.request :url_encoded

  f.adapter :net_http
end

file_io = Faraday::UploadIO.new('photo.jpg', 'image/jpeg')

conn.post('http://example.com/profile', file: file_io)
```

How to Send Custom HTTP Headers

You can send custom headers with an HTTP request.

This helps you send extra data with your request, including cookies, user-agent, and caching information.

Here's how:

```
Faraday.new('http://example.com', headers: { 'User-Agent' => 'test' }).get
```

I'm creating a Faraday object (using new), then calling get on it.

This doesn't work if you call get directly.

Choosing The Best Ruby HTTP Client

There are many HTTP clients available in Ruby.

Authentication using digest

Green Field

We start with a fresh Rails application:

```
$ rails new shop
$ cd shop
```

Later we are going to redirect to root. So we start with creating an empty root page:

```
$ rails g controller home index
```

Please add the following code to `config/routes.rb` :

```
Rails.application.routes.draw do
  root ‘home#index’
end
```

And some content for that page in the file `app/views/home/index.html.erb` :

```
<p id=”notice”><%= notice %></p>
<h1>Example</h1>
<p>Lorem ipsum …</p>
```

Password Digest

Obviously we do not store the clear text password in the database but a digest of it. For that we need to activate the `bcrypt` gem in the file `Gemfile`:

```
# Use ActiveModel has_secure_password
gem ‘bcrypt’, ‘~> 3.1.7’
```

And run bundle afterwords:

```
$ bundle
```

User Model

Now we create a `User` scaffold. Feel free to add any additional fields you might need (e.g. first_name, last_name). I just use `email:uniq` to store the email address (and create an unique database index) and `password:digest` to create a `password_digest` field in the new `users` table.

```
$ rails g scaffold User email:uniq password:digest
$ rails db:migrate
```

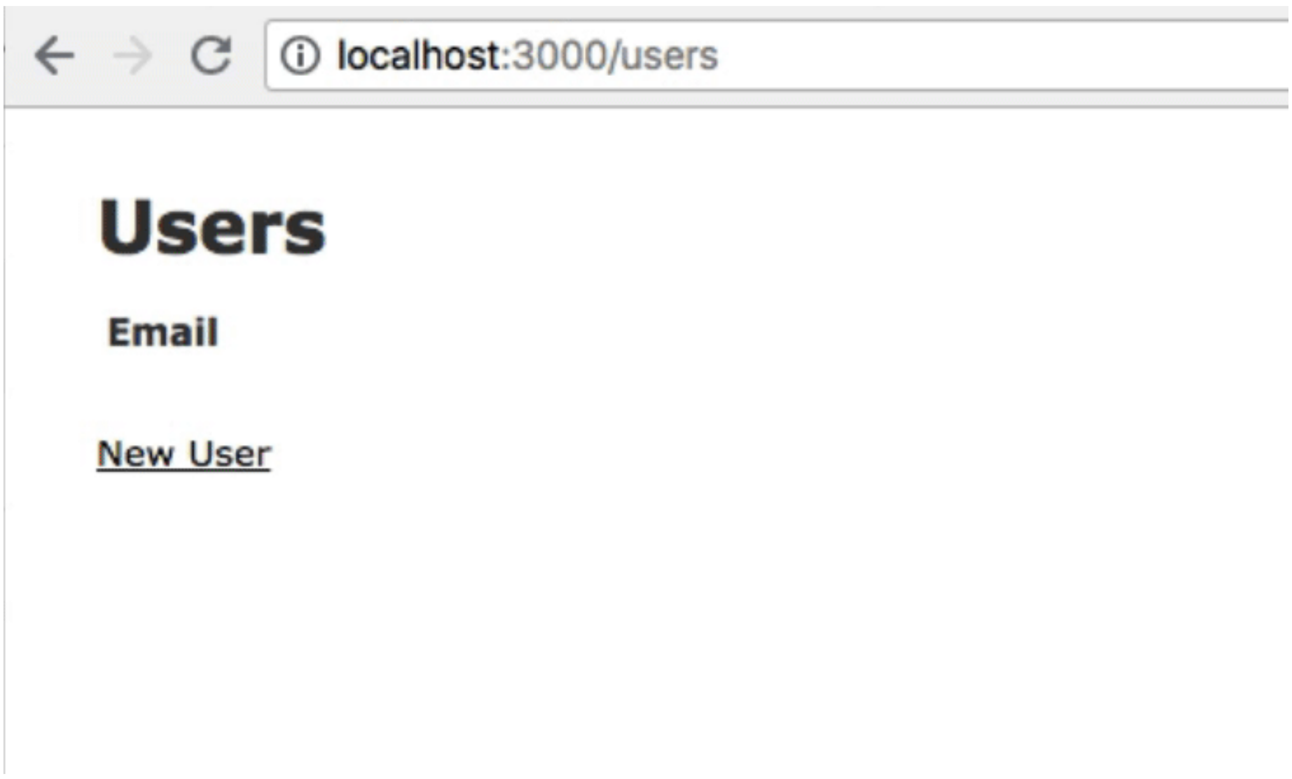
The `digest` part puts some Rails magic into action. The Rails generator creates a `password_digest` field in the table and asks for an additional `password_confirmation` in the form and the controllers `user_params` without you having to do anything extra. `has_secure_password` in the model takes care of encrypting the password and provides the`authenticate` method to authenticate with that password.

Before a first test we need to add some validations in `app/models/user.rb` to make sure that we have an email address and that it is unique:

```
class User < ApplicationRecord
  has_secure_password
  validates :email, presence: true, uniqueness: true
end
```

Now we can fire up Rails and create a new user in the browser:

```
$ rails s
```



localhost:3000/users

Users

Email

New User

Screencast: Create a new user at http://localhost:3000/users/new

Let’s just check how Rails stores the password digest in the table:

```
$ rails c
Running via Spring preloader in process 3204
Loading development environment (Rails 5.2.1)
>> User.first
User Load (0.2ms) SELECT “users”.* FROM “users”
ORDER BY “users”.”id” ASC LIMIT ? [[“LIMIT”, 1]]
=> #<User id: 1, email: “sw@wintermeyer-
consulting.de”, password_digest:
“$2a$10$t6Q2R.N5fevFjhL/
W1X.EulEJQ8TDWIzCvHpbDrAtQo…”, created_at: “2018-
09-18 12:13:26”, updated_at: “2018-09-18
12:13:26”>
```

So only the digest is saved. Everything is secure.

But we still need to create some sort of login to actually use it.

Sessions

When a user logs in he/she creates a new session. When the same user logs out that session gets destroyed. Therefor we create a sessions controller with three actions:

```
$ rails g controller sessions new create destroy
```

We put the following code into `app/controllers/sessions_controller.rb`:

```
class SessionsController < ApplicationController
  def new
  end
  def create
    user = User.find_by_email(params[:email])
    if user && user.authenticate(params[:password])
      session[:user_id] = user.id
      redirect_to root_url, notice: "Logged in!"
    else
      flash.now[:alert] = "Email or password is invalid"
      render "new"
    end
  end
  def destroy
    session[:user_id] = nil
    redirect_to root_url, notice: "Logged out!"
  end
end
```

As you can see we use `session[:user_id]` to store the logged in user id. In case you haven't worked with sessions yet have a look at https://guides.rubyonrails.org/security.html#sessions
We need to put this code for the form in `app/views/sessions/new.html.erb`:

```
<p id=”alert”><%= alert %></p>
<h1>Login</h1>
<%= form_tag sessions_path do |form| %>
  <div class=”field”>
    <%= label_tag :email %>
    <%= text_field_tag :email %>
  </div>
  <div class=”field”>
    <%= label_tag :password %>
    <%= password_field_tag :password %>
  </div>
  <div class=”actions”>
    <%= submit_tag “Login” %>
  </div>
<% end %>
```

Routes

The routes are a bit cumbersome. But we can fix this with this code in `config/routes.rb` :

```
Rails.application.routes.draw do
  root ‘home#index’
  resources :users
  resources :sessions, only: [:new, :create, :destroy]
  get ‘signup’, to: ‘users#new’, as: ‘signup’
  get ‘login’, to: ‘sessions#new’, as: ‘login’
  get ‘logout’, to: ‘sessions#destroy’, as: ‘logout’
end
```

Now a user can use http://localhost:3000/login to login and http://localhost:3000/logout to logout. Much easier for everybody.

current_user

In most Rails applications the logged in user is available with a `current_user` helper. This come handy too if you want to use an authorization gem like cancancan. The most popular way to add this functionality is this code in `app/controllers/application_controller.rb`:

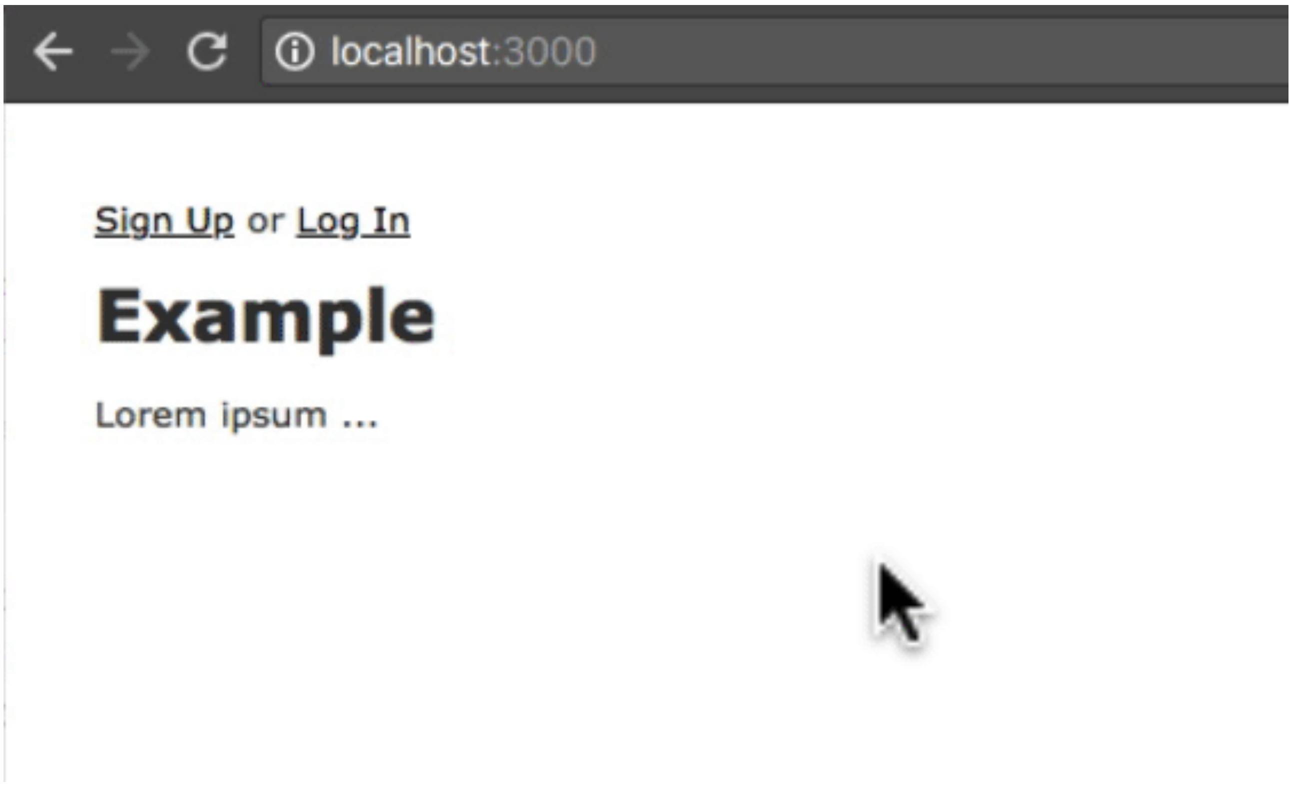
```
class ApplicationController < ActionController::Base
  helper_method :current_user
  def current_user
    if session[:user_id]
      @current_user ||= User.find(session[:user_id])
    else
      @current_user = nil
    end
  end
end
```

To put it into use we change the content of `app/views/home/index.html.erb`:

```
<% if current_user %>
  Logged in as <%= current_user.email %>.
  <%= link_to “Log Out”, logout_path %>
<% else %>
  <%= link_to “Sign Up”, signup_path %> or
  <%= link_to “Log In”, login_path %>
<% end %>
<p id=”notice”><%= notice %></p>
<h1>Example</h1>
<p>Lorem ipsum …</p>
```

The End

And here is the screencast where I log in with my account and log out afterwards:



Sign Up or Log In

Example

Lorem ipsum ...

Screencast: Log in and Log out