DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Active Record Validations

This guide teaches you how to validate the state of objects before they go into the database using Active Record's validations feature.

After reading this guide, you will know:

- How to use the built-in Active Record validation helpers.
- How to create your own custom validation methods.
- How to work with the error messages generated by the validation process.

Validations Overview

Here's an example of a very simple validation:

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

```ruby
irb> Person.create(name: "John Doe").valid?
=> true
irb> 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.

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 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.

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 Active Record class:

```
class Person < ApplicationRecord
end
```

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

```
irb> p = Person.new(name: "John Doe")
=> #<Person id: nil, name: "John Doe", created_at: nil, updated_at: nil>

irb> p.new_record?
=> true

irb> p.save
=> true

irb> 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.

CAUTION: 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` returns the object.

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`
- `insert`
- `insert!`
- `insert_all`
- `insert_all!`
- `toggle!`
- `touch`
- `touch_all`
- `update_all`
- `update_attribute`
- `update_column`
- `update_columns`
- `update_counters`
- `upsert`
- `upsert_all`

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)`

`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
```

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

After Active Record has performed validations, any errors found can be accessed through the `errors` 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
```

```
irb> p = Person.new
=> #<Person id: nil, name: nil>
irb> p.errors.size
=> 0

irb> p.valid?
=> false
irb> p.errors.objects.first.full_message
=> "Name can't be blank"

irb> p = Person.create
=> #<Person id: nil, name: nil>
irb> p.errors.objects.first.full_message
=> "Name can't be blank"

irb> p.save
=> false

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

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

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

`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 error messages 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
```

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

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

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 is added to the object's `errors` collection, and this 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.

`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 in a 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 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.

`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"*.

`comparison`

This check will validate a comparison between any two comparable values. The validator requires a compare option be supplied. Each option accepts a value, proc, or symbol. Any class that includes Comparable can be compared.

```
class Promotion < ApplicationRecord
  validates :end_date, comparison: { greater_than: :start_date }
end
```

These options are all supported:

- `: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}"*.

`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"*.

`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"*.

`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*".

`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 customize 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 custom message or use `presence: true` instead. When `:in` or `:within` have a lower limit of 1, you should either provide a custom message or call `presence` prior to `length`.

`numericality`

This helper validates that your attributes have only numeric values. By default, it will match an optional sign followed by an integer or floating point number.

To specify that only integer numbers are allowed, 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`. `Float`s are casted to `BigDecimal` using the column's precision value or 15.

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

The default error message for `:only_integer` is "*must be an integer*".

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}*".
- `:in` - Specifies the value must be in the supplied range. The default error message for this option is "*must be in %{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*".

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

The default error message when no options are specified is "*is not a number*".

`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. This way, it is not only checked that the foreign key is not empty but also that the referenced object exists.

```ruby
class Supplier < ApplicationRecord
  has_one :account
  validates :account, presence: true
end
```

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

NOTE: If you want to ensure that the association it is both present and valid, you also need to use `validates_associated`.

```ruby
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:

```ruby
validates :boolean_field_name, inclusion: [true, false]
validates :boolean_field_name, exclusion: [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.

`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.

```ruby
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.

```ruby
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:

```ruby
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*".

`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, case insensitive, or respects default database collation. This option defaults to respects default database collation.

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

WARNING. Note that some databases are configured to perform case-insensitive searches anyway.

The default error message is "*has already been taken*".

`validates_associated`

You should use this helper when your model has associations that always need to be validated. Every time you try to save your object, `valid?` will be called on 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.

CAUTION: 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.

`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.add :base, "This person is evil"
    end
  end
end

class Person < ApplicationRecord
  validates_with GoodnessValidator
end
```

NOTE: 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.add :base, "This person is evil"
    end
  end
```

```ruby
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:

```ruby
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_methods?
      @person.errors.add :base, "This person is evil"
    end
  end

  # ...
end
```

`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.

```ruby
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 to the model, therefore making it invalid.

Common Validation Options

These are common validation options:

`: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.

`: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
```

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

`: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}
  # are 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 already taken."
      end
    }
end
```

`: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 contexts. Custom contexts need to be triggered explicitly by passing the 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
```

```
irb> person = Person.new(age: 'thirty-three')
irb> person.valid?
=> true
irb> person.valid?(:account_setup)
=> false
irb> person.errors.messages
=> {:email=>["has already been taken"], :age=>["is not a number"]}
```

`person.valid?(:account_setup)` executes both the validations without saving the model. `person.save(context: :account_setup)` validates `person` in the `account_setup` context before saving.

When triggered by an explicit context, validations are run for that context, as well as any validations *without* a context.

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

```
irb> person = Person.new
irb> person.valid?(:account_setup)
=> false
irb> person.errors.messages
=> {:email=>["has already been taken"], :age=>["is not a number"], :name=>["can't be
blank"]}
```

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
```

```
irb> 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
```

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

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.

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
```

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

It is 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
```

As `Lambdas` are a type of `Proc`, they can also be used to write inline conditions in a shorter way.

```
validates :password, confirmation: true, unless: -> { password.blank? }
```

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?`

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`.

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.

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.start_with? 'X'
      record.errors.add :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 =~ URI::MailTo::EMAIL_REGEXP
      record.errors.add 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.

Custom Methods

You can also create methods that verify the state of your models and add errors to the `errors` collection when they are invalid. You must then register these methods by using the `validate` 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
```

Working with Validation Errors

The `valid?` and `invalid?` methods only provide a summary status on validity. However you can dig deeper into each individual error by using various methods from the `errors` collection.

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.

`errors`

The gateway through which you can drill down into various details of each error.

This returns an instance of the class `ActiveModel::Errors` containing all errors, each error is represented by an `ActiveModel::Error` object.

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

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

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

`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
```

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

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

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

`errors.where` and error object

Sometimes we may need more information about each error beside its message. Each error is encapsulated as an `ActiveModel::Error` object, and `where` method is the most common way of access.

`where` returns an array of error objects, filtered by various degree of conditions.

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

```
irb> person = Person.new
irb> person.valid?
=> false

irb> person.errors.where(:name)
=> [ ... ] # all errors for :name attribute

irb> person.errors.where(:name, :too_short)
=> [ ... ] # :too_short errors for :name attribute
```

You can read various information from these error objects:

```
irb> error = person.errors.where(:name).last

irb> error.attribute
=> :name
irb> error.type
=> :too_short
irb> error.options[:count]
=> 3
```

You can also generate the error message:

```
irb> error.message
=> "is too short (minimum is 3 characters)"
irb> error.full_message
=> "Name is too short (minimum is 3 characters)"
```

The `full_message` method generates a more user-friendly message, with the capitalized attribute name prepended.

`errors.add`

The `add` method creates the error object by taking the `attribute`, the error `type` and additional options hash. This is useful for writing your own validator.

```
class Person < ApplicationRecord
  validate do |person|
    errors.add :name, :too_plain, message: "is not cool enough"
```

```
  end
end
```

```
irb> person = Person.create
irb> person.errors.where(:name).first.type
=> :too_plain
irb> person.errors.where(:name).first.full_message
=> "Name is not cool enough"
```

`errors[:base]`

You can add errors that are related to the object's state as a whole, instead of being related to a specific attribute. You can add errors to `:base` when you want to say that the object is invalid, no matter the values of its attributes.

```
class Person < ApplicationRecord
  validate do |person|
    errors.add :base, :invalid, message: "This person is invalid because ..."
  end
end
```

```
irb> person = Person.create
irb> person.errors.where(:base).first.full_message
=> "This person is invalid because ..."
```

`errors.clear`

The `clear` method is used when you intentionally want to clear 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
```

```
irb> person = Person.new
irb> person.valid?
=> false
irb> person.errors.empty?
=> false

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

irb> person.save
=> false
```

```
irb> person.errors.empty?
=> false
```

`errors.size`

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

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

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

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

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 fails.

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, you can 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.each do |error| %>
        <li><%= error.full_message %></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.