Active Model Basics

January 13, 2015

This guide should provide you with all you need to get started using model classes. Active Model allows for Action Pack helpers to interact with plain Ruby objects. Active Model also helps build custom ORMs for use outside of the Rails framework.

After reading this guide, you will be able to add to plain Ruby objects:

- The ability to behave like an Active Record model.
- Callbacks and validations like Active Record.
- Serializers.
- Integration with the Rails internationalization (i18n) framework.

1 Introduction

Active Model is a library containing various modules used in developing classes that need some features present on Active Record. Some of these modules are explained below.

1.1 Attribute Methods

The `ActiveModel::AttributeMethods` module can add custom prefixes and suffixes on methods of a class. It is used by defining the prefixes and suffixes and which methods on the object will use them.

```
class Person
  include ActiveModel::AttributeMethods

  attribute_method_prefix 'reset_'
  attribute_method_suffix '_highest?'
  define_attribute_methods 'age'

  attr_accessor :age

  private
    def reset_attribute(attribute)
      send("#{attribute}=", 0)
    end

    def attribute_highest?(attribute)
```

```
      send(attribute) > 100
    end
end

person = Person.new
person.age = 110
person.age_highest?  # => true
person.reset_age     # => 0
person.age_highest?  # => false
```

1.2 Callbacks

`ActiveModel::Callbacks` gives Active Record style callbacks. This provides an ability to define callbacks which run at appropriate times. After defining callbacks, you can wrap them with before, after and around custom methods.

```
class Person
  extend ActiveModel::Callbacks

  define_model_callbacks :update

  before_update :reset_me

  def update
    run_callbacks(:update) do
      # This method is called when update is called on an object.
    end
  end

  def reset_me
    # This method is called when update is called on an object as a before_update callback
is defined.
  end
end
```

1.3 Conversion

If a class defines `persisted?` and `id` methods, then you can include the `ActiveModel::Conversion` module in that class and call the Rails conversion methods on objects of that class.

```
class Person
  include ActiveModel::Conversion

  def persisted?
    false
  end
```

```
  def id
    nil
  end
end

person = Person.new
person.to_model == person  # => true
person.to_key              # => nil
person.to_param            # => nil
```

1.4 Dirty

An object becomes dirty when it has gone through one or more changes to its attributes and has not been saved. `ActiveModel::Dirty` gives the ability to check whether an object has been changed or not. It also has attribute based accessor methods. Let's consider a Person class with attributes `first_name` and `last_name`:

```
class Person
  include ActiveModel::Dirty
  define_attribute_methods :first_name, :last_name

  def first_name
    @first_name
  end

  def first_name=(value)
    first_name_will_change!
    @first_name = value
  end

  def last_name
    @last_name
  end

  def last_name=(value)
    last_name_will_change!
    @last_name = value
  end

  def save
    # do save work...
    changes_applied
  end
end
```

1.4.1 Querying object directly for its list of all changed attributes.

```
person = Person.new
person.changed? # => false

person.first_name = "First Name"
person.first_name # => "First Name"

# returns if any attribute has changed.
person.changed? # => true

# returns a list of attributes that have changed before saving.
person.changed # => ["first_name"]

# returns a hash of the attributes that have changed with their original values.
person.changed_attributes # => {"first_name"=>nil}

# returns a hash of changes, with the attribute names as the keys, and the values will be
an array of the old and new value for that field.
person.changes # => {"first_name"=>[nil, "First Name"]}
```

1.4.2 Attribute based accessor methods Track whether the particular attribute has been changed or not.

```
# attr_name_changed?
person.first_name # => "First Name"
person.first_name_changed? # => true
```

Track what was the previous value of the attribute.

```
# attr_name_was accessor
person.first_name_was # => nil
```

Track both previous and current value of the changed attribute. Returns an array if changed, else returns nil.

```
# attr_name_change
person.first_name_change # => [nil, "First Name"]
person.last_name_change # => nil
```

1.5 Validations

`ActiveModel::Validations` module adds the ability to validate class objects like in Active Record.

```
class Person
  include ActiveModel::Validations
```

```
  attr_accessor :name, :email, :token

  validates :name, presence: true
  validates_format_of :email, with: /\A([^\s]+)((?:[-a-z0-9]\.)[a-z]{2,})\z/i
  validates! :token, presence: true
end

person = Person.new
person.token = "2b1f325"
person.valid?                          # => false
person.name = 'vishnu'
person.email = 'me'
person.valid?                          # => false
person.email = 'me@vishnuatrai.com'
person.valid?                          # => true
person.token = nil
person.valid?                          # => raises ActiveModel::StrictValidationFailed
```

1.6 Naming

`ActiveModel::Naming` adds a number of class methods which make the naming and routing easier to manage. The module defines the `model_name` class method which will define a number of accessors using some `ActiveSupport::Inflector` methods.

```
class Person
  extend ActiveModel::Naming
end

Person.model_name.name                 # => "Person"
Person.model_name.singular             # => "person"
Person.model_name.plural               # => "people"
Person.model_name.element              # => "person"
Person.model_name.human                # => "Person"
Person.model_name.collection           # => "people"
Person.model_name.param_key            # => "person"
Person.model_name.i18n_key             # => :person
Person.model_name.route_key            # => "people"
Person.model_name.singular_route_key   # => "person"
```

1.7 Model

`ActiveModel::Model` adds the ability to a class to work with Action Pack and Action View right out of the box.

```
class EmailContact
  include ActiveModel::Model
```

```
  attr_accessor :name, :email, :message
  validates :name, :email, :message, presence: true

  def deliver
    if valid?
      # deliver email
    end
  end
end
```

When including `ActiveModel::Model` you get some features like:

- model name introspection
- conversions
- translations
- validations

It also gives you the ability to initialize an object with a hash of attributes, much like any Active Record object.

```
email_contact = EmailContact.new(name: 'David',
                                 email: 'david@example.com',
                                 message: 'Hello World')
email_contact.name       # => 'David'
email_contact.email      # => 'david@example.com'
email_contact.valid?     # => true
email_contact.persisted? # => false
```

Any class that includes `ActiveModel::Model` can be used with `form_for`, `render` and any other Action View helper methods, just like Active Record objects.

1.8 Serialization

`ActiveModel::Serialization` provides a basic serialization for your object. You need to declare an attributes hash which contains the attributes you want to serialize. Attributes must be strings, not symbols.

```
class Person
  include ActiveModel::Serialization

  attr_accessor :name

  def attributes
    {'name' => nil}
  end
end
```

Now you can access a serialized hash of your object using the `serializable_hash`.

```
person = Person.new
person.serializable_hash   # => {"name"=>nil}
person.name = "Bob"
person.serializable_hash   # => {"name"=>"Bob"}
```

1.8.1 ActiveModel::Serializers Rails provides two serializers `ActiveModel::Serializers::JSON` and `ActiveModel::Serializers::Xml`. Both of these modules automatically include the `ActiveModel::Serialization`.

1.8.1.1 ActiveModel::Serializers::JSON

To use the `ActiveModel::Serializers::JSON` you only need to change from `ActiveModel::Serialization` to `ActiveModel::Serializers::JSON`.

```
class Person
  include ActiveModel::Serializers::JSON

  attr_accessor :name

  def attributes
    {'name' => nil}
  end
end
```

With the `as_json` you have a hash representing the model.

```
person = Person.new
person.as_json # => {"name"=>nil}
person.name = "Bob"
person.as_json # => {"name"=>"Bob"}
```

From a JSON string you define the attributes of the model. You need to have the `attributes=` method defined on your class:

```
class Person
  include ActiveModel::Serializers::JSON

  attr_accessor :name

  def attributes=(hash)
    hash.each do |key, value|
      send("#{key}=", value)
    end
  end

  def attributes
    {'name' => nil}
  end
end
```

Now it is possible to create an instance of person and set the attributes using `from_json`.

```
json = { name: 'Bob' }.to_json
person = Person.new
person.from_json(json) # => #<Person:0x00000100c773f0 @name="Bob">
person.name            # => "Bob"
```

1.8.1.2 ActiveModel::Serializers::Xml

To use the `ActiveModel::Serializers::Xml` you only need to change from `ActiveModel::Serialization` to `ActiveModel::Serializers::Xml`.

```
class Person
  include ActiveModel::Serializers::Xml

  attr_accessor :name

  def attributes
    {'name' => nil}
  end
end
```

With the `to_xml` you have a XML representing the model.

```
person = Person.new
person.to_xml # => "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<person>\n
<name nil=\"true\"/>\n</person>\n"
person.name = "Bob"
person.to_xml # => "<?xml version=\"1.0\" encoding=\"UTF-8\"?>\n<person>\n
<name>Bob</name>\n</person>\n"
```

From a XML string you define the attributes of the model. You need to have the `attributes=` method defined on your class:

```
class Person
  include ActiveModel::Serializers::Xml

  attr_accessor :name

  def attributes=(hash)
    hash.each do |key, value|
      send("#{key}=", value)
    end
  end

  def attributes
    {'name' => nil}
  end
end
```

Now it is possible to create an instance of person and set the attributes using `from_xml`.

```
xml = { name: 'Bob' }.to_xml
person = Person.new
person.from_xml(xml) # => #<Person:0x00000100c773f0 @name="Bob">
person.name          # => "Bob"
```

1.9 Translation

`ActiveModel::Translation` provides integration between your object and the Rails internationalization (i18n) framework.

```
class Person
  extend ActiveModel::Translation
end
```

With the `human_attribute_name` you can transform attribute names into a more human format. The human format is defined in your locale file.

- config/locales/app.pt-BR.yml

```
  pt-BR:
    activemodel:
      attributes:
        person:
          name: 'Nome'
```

```
Person.human_attribute_name('name') # => "Nome"
```

1.10 Lint Tests

`ActiveModel::Lint::Tests` allow you to test whether an object is compliant with the Active Model API.

- app/models/person.rb

```
class Person
  include ActiveModel::Model

end
```

- test/models/person_test.rb

```
require 'test_helper'

class PersonTest < ActiveSupport::TestCase
  include ActiveModel::Lint::Tests
```

```
      def setup
        @model = Person.new
      end
    end

$ rake test

Run options: --seed 14596

# Running:

......

Finished in 0.024899s, 240.9735 runs/s, 1204.8677 assertions/s.

6 runs, 30 assertions, 0 failures, 0 errors, 0 skips
```

An object is not required to implement all APIs in order to work with Action Pack. This module only intends to provide guidance in case you want all features out of the box.

1.11 SecurePassword

`ActiveModel::SecurePassword` provides a way to securely store any password in an encrypted form. On including this module, a `has_secure_password` class method is provided which defines an accessor named `password` with certain validations on it.

1.11.1 Requirements `ActiveModel::SecurePassword` depends on the `bcrypt`, so include this gem in your Gemfile to use `ActiveModel::SecurePassword` correctly. In order to make this work, the model must have an accessor named `password_digest`. The `has_secure_password` will add the following validations on the `password` accessor:

1. Password should be present.
2. Password should be equal to its confirmation.
3. This maximum length of a password is 72 (required by `bcrypt` on which ActiveModel::SecurePassword depends)

1.11.2 Examples

```
class Person
  include ActiveModel::SecurePassword
  has_secure_password
  attr_accessor :password_digest
end

person = Person.new
```

```
# When password is blank.
person.valid? # => false

# When the confirmation doesn't match the password.
person.password = 'aditya'
person.password_confirmation = 'nomatch'
person.valid? # => false

# When the length of password, exceeds 72.
person.password = person.password_confirmation = 'a' * 100
person.valid? # => false

# When all validations are passed.
person.password = person.password_confirmation = 'aditya'
person.valid? # => true
```

2 Feedback

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

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

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

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

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