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

Active Model Basics

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 know:

- How an Active Record model behaves.
- How Callbacks and validations work.
- How serializers work.
- How Active Model integrates with the Rails internationalization (i18n) framework.

What is Active Model?

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.

API

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

```
class EmailContact
  include ActiveModel::API

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

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

When including `ActiveModel::API` 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.

```
irb> email_contact = EmailContact.new(name: 'David', email: 'david@example.com',
message: 'Hello World')
irb> email_contact.name
=> "David"
irb> email_contact.email
```

```
=> "david@example.com"
irb> email_contact.valid?
=> true
irb> email_contact.persisted?
=> false
```

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

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
```

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

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
```

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
```

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

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
```

Querying object directly for its list of all changed attributes.

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

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

# Returns true if any of the attributes have unsaved changes.
irb> person.changed?
=> true

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

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

# Returns a Hash of changes, with the attribute names as the keys, and the values as
an array of the old and new values for that field.
```

```
irb> person.changes
=> {"first_name"=>[nil, "First Name"]}
```

Attribute-based accessor methods

Track whether the particular attribute has been changed or not.

```
irb> person.first_name
=> "First Name"

# attr_name_changed?
irb> person.first_name_changed?
=> true
```

Track the previous value of the attribute.

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

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

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

Validations

The `ActiveModel::Validations` module adds the ability to validate 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
```

```
irb> person = Person.new
irb> person.token = "2b1f325"
irb> person.valid?
=> false
irb> person.name = 'vishnu'
irb> person.email = 'me'
irb> person.valid?
```

```
=> false
irb> person.email = 'me@vishnuatrai.com'
irb> person.valid?
=> true
irb> person.token = nil
irb> person.valid?
ActiveModel::StrictValidationFailed
```

Naming

`ActiveModel::Naming` adds several class methods which make naming and routing easier to manage. The module defines the `model_name` class method which will define several 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"
```

Model

`ActiveModel::Model` allows implementing models similar to `ActiveRecord::Base`.

```
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 all the features from `ActiveModel::API`.

Serialization

`ActiveModel::Serialization` provides 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` method.

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

ActiveModel::Serializers

Active Model also provides the `ActiveModel::Serializers::JSON` module for JSON serializing / deserializing. This module automatically includes the previously discussed `ActiveModel::Serialization` module.

ActiveModel::Serializers::JSON

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

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

  attr_accessor :name

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

The `as_json` method, similar to `serializable_hash`, provides a Hash representing the model.

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

You can also define the attributes for a model from a JSON string. However, you need to define the `attributes=` method 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 attributes using `from_json`.

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

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` method, you can transform attribute names into a more human-readable format. The human-readable format is defined in your locale file(s).

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

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

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

Lint Tests

`ActiveModel::Lint::Tests` allows 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

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

```
$ bin/rails 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 guide in case you want all features out of the box.

SecurePassword

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

Requirements

`ActiveModel::SecurePassword` depends on `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

`XXX_digest`. Where `XXX` is the attribute name of your desired password. The following validations are added automatically:

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

Examples

```
class Person
  include ActiveModel::SecurePassword
  has_secure_password
  has_secure_password :recovery_password, validations: false

  attr_accessor :password_digest, :recovery_password_digest
end
```

```
irb> person = Person.new

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

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

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

# When only password is supplied with no password_confirmation.
irb> person.password = 'aditya'
irb> person.valid?
=> true

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

irb> person.recovery_password = "42password"

irb> person.authenticate('aditya')
=> #<Person> # == person
irb> person.authenticate('notright')
=> false
```

```
irb> person.authenticate_password('aditya')
=> #<Person> # == person
irb> person.authenticate_password('notright')
=> false

irb> person.authenticate_recovery_password('42password')
=> #<Person> # == person
irb> person.authenticate_recovery_password('notright')
=> false

irb> person.password_digest
=> "$2a$04$gF8RfZdoXHvyTjHhiU4ZsO.kQqV9oonYZu31PRE4hLQn3xM2qkpIy"
irb> person.recovery_password_digest
=> "$2a$04$iOfhwahFymCs5weB3BNH/uXkTG65HR.qpW.bNhEjFP3ftli3o5DQC"
```