validations.rdoc
doc/validations.rdoc
Last Update: 2011-07-16 11:31:03 -0700

Model Validations

This guide is based on
guides.rubyonrails.org/activerecord_validations_callbacks.html

Overview

This guide is designed to teach you how to use Sequel::Model's validation support. It attempts to explain how Sequel's validation support works, what validations are useful for, and how to use the `validation_helpers` plugin to add specific types of validations to your models.

Why Validations?

Validations are primarily useful for associating error messages to display to the user with specific attributes on the model. It is also possible to use them to enforce data integrity for model instances, but that's not a recommended use unless the only way to modify the database is through model instances, or you have complex data integrity requirements that aren't possible to specify via database-level constraints.

Data Integrity

Data integrity is best handled by the database itself. For example, if you have a date column that should never contain a NULL value, the column should be specified in the database as NOT NULL. If you have an integer column that should only have values from 1 to 10, there should be a CHECK constraint that ensures that the value of that column is between 1 and 10. And if you have a varchar column where the length of the entries should be between 2 and 255, you should be setting the size of the varchar column to 255, and using a CHECK constraint to ensure that all values have at least two characters.

Unfortunately, sometimes there are situations where that is not possible. For example, if you are using MySQL and don't have control over the database configuration, it's possible that if you attempt to insert a string with 300 characters into a varchar(255) field, that MySQL will just silently truncate it for you, instead of raising an error. In that case, it may be necessary to use a

model validation to enforce the database integrity.

Also, in some cases you may have data integrity requirements that are difficult to enforce via database constraints, especially if you are targetting multiple database types.

Finally, validations are generally easier to write than database constraints, so if data integrity isn't of great importance, using validations to provide minimal data integrity is probably fine.

Usage

Regardless of whether you are using validations for data integrity or just for error messages, the usage is the same. Whenever you attempt to save a model instance, before sending the INSERT or UPDATE query to the database, `Sequel::Model` will attempt to validate the instance by calling `validate`. If `validate` does not add any errors to the object, the object is considered valid, and `valid?` will return true. If `validate` adds any errors to the object, `valid?` will return false, and the save will either raise a `Sequel::ValidationFailed` exception (the default), or return nil (if `raise_on_save_failure` is false).

By validating the object before sending the database query, Sequel attempts to ensure that invalid objects are not saved in the database. However, if you are not enforcing the same validations in the database via constraints, it's possible that invalid data can get added to the database via some other method. This leads to odd cases such as retrieving a model object from the database, not making any changes to it, attempting to save it, and having the save raise an error.

Skipping Validations

`Sequel::Model` uses the `save` method to save model objects, and all saving of model objects passes through the `save` method. This means that all saving of model objects goes through the validation process.

The only way to skip validations when saving a model object is to pass the `:validate => false` option to `save`. If you use that option, `save` will not attempt to validate the object before saving it.

Note that it's always possible to update the instance's database row without using `save`, by using a dataset to update it. Validations will only be run if you call `save` on the model object, or another model method that calls `save`. For example, the `create` class method instantiates a new instance of the model, and then calls `save`, so it validates the object. However, the `insert` class method is a

dataset method that just inserts the raw hash into the database, so it doesn't validate the object.

`valid?` and `validate`

`Sequel::Model` uses the `valid?` method to check whether or not a model instance is valid. This method should not be overridden. Instead, the `validate` method should be overridden to add validations to the model:

```
class Album < Sequel::Model
  def validate
    super
    errors.add(:name, 'cannot be empty') if !name || name.empty?
  end
end
Album.new.valid? # false
Album.new(:name=>'').valid? # false
Album.new(:name=>'RF').valid? # true
```

If the `valid?` method returns false, you can call the `errors` method to get an instance of `Sequel::Model::Errors` describing the errors on the model:

```
a = Album.new
# => #<Album @values={}>
a.valid?
# => false
a.errors
# => {:name=>["cannot be empty"]}
```

You may notice that the `errors` method appears to return a hash. That's because `Sequel::Model::Errors` is a subclass of Hash.

Note that calling the `errors` method before the `valid?` method will result in an `errors` being empty:

```
Album.new.errors
# => {}
```

So just remember that you shouldn't check `errors` until after you call `valid?`.

`Sequel::Model::Errors` has some helper methods that make it easy to get an array of all of the instance's errors, or for checking for errors on a specific attribute. These will be covered later in this guide.

`validation_helpers`

While `Sequel::Model` does provide a validations framework, it does not define any built-in validation helper methods that you can call. However, Sequel ships

with a plugin called `validation_helpers` that handles most basic validation needs. So instead of specifying validations like this:

```
class Album < Sequel::Model
  def validate
    super
    errors.add(:name, 'cannot be empty') if !name || name.empty?
    errors.add(:name, 'is already taken') if name && new? && Album[:name=>name]
    errors.add(:website, 'cannot be empty') if !website || website.empty?
    errors.add(:website, 'is not a valid URL') unless website =~ /\Ahttps?:\/\//
  end
end
```

You can call simple methods such as:

```
class Album < Sequel::Model
  plugin :validation_helpers
  def validate
    super
    validates_presence [:name, :website]
    validates_unique :name
    validates_format /\Ahttps?:\/\//, :website, :message=>'is not a valid URL'
  end
end
```

Other than `validates_unique`, which has it's own API, the methods defined by `validation_helpers` have one of the following two APIs:

`(atts, opts={})`	For methods such as `validates_presence`, which do not take an additional argument.
`(arg, atts, opts={})`	For methods such as `validates_format`, which take an additional argument.

For both of these APIs, `atts` is either a column symbol or array of column symbols, and `opts` is an optional options hash.

The following methods are provided by `validation_helpers`:

`validates_presence`

This is probably the most commonly used helper method, which checks if the specified attributes are not blank. In general, if an object responds to `blank?`, it calls the method to determine if the object is blank. Otherwise, nil is considered blank, empty strings or strings that just contain whitespace are blank, and objects that respond to `empty?` and return true are considered blank.

All other objects are considered non-blank for the purposes of `validates_presence`. This means that `validates_presence` is safe to use on boolean columns where you want to ensure that either true or false is used, but not NULL.

```
class Album < Sequel::Model
  def validate
    validates_presence [:name, :website, :debut_album]
  end
end
```

validates_format

`validates_format` is used to ensure that the string value of the specified attributes matches the specified regular expression. It's useful for checking that fields such as email addresses, URLs, UPC codes, ISBN codes, and the like, are in a specific format. It can also be used to validate that only certain characters are used in the string.

```
class Album < Sequel::Model
  def validate
    validates_format /\A\d\d\d-\d-\d{7}-\d-\d\z/, :isbn
    validates_format /\a[0-9a-zA-Z:' ]+\z/, :name
  end
end
```

validates_exact_length, validates_min_length, validates_max_length, validates_length_range

These methods all deal with ensuring that the length of the specified attribute matches the criteria specified by the first argument to the method. `validates_exact_length` is for checking that the length of the attribute is equal to that value, `validates_min_length` is for checking that the length of the attribute is greater than or equal to that value, `validates_max_length` is for checking that the length of the attribute is less than or equal to that value, and `validates_length_range` is for checking that the length of the attribute falls in the value, which should be a range or another object that responds to `include?`.

```
class Album < Sequel::Model
  def validate
    validates_exact_length 17, :isbn
    validates_min_length 3, :name
    validates_max_length 100, :name
    validates_length_range 3..100, :name
  end
end
```

validates_integer, validates_numeric

These methods check that the specified attributes can be valid integers or valid floats. `validates_integer` tests the attribute value using `Kernel.Integer` and `validates_numeric` tests the attribute using `Kernel.Float`. If the Kernel methods raise an exception, the validation fails, otherwise it succeeds.

```
class Album < Sequel::Model
  def validate
    validates_integer :copies_sold
    validates_numeric :replaygain
  end
end
```

validates_includes

`validates_includes` checks that the specified attributes are included in the first argument to the method, which is usually an array, but can be any object that responds to `include?`.

```
class Album < Sequel::Model
  def validate
    validates_includes [1, 2, 3, 4, 5], :rating
  end
end
```

validates_type

`validates_type` checks that the specified attributes are instances of the class specified in the first argument. The class can be specified as the class itself, or as a string or symbol with the class name:

```
class Album < Sequel::Model
  def validate
    validates_type String, [:name, :website]
    validates_type :Artist, :artist
  end
end
```

validates_not_string

`validates_not_string` is a special validation designed to be used with the `raise_on_typecast_failure = false` setting. By default, Sequel raises errors when typecasting fails, immediately when you try to set the value on the object:

```
album = Album.new
album.copies_sold = 'banana' # raises Sequel::InvalidValue
```

The `raise_on_typecast_failure = false` setting tells Sequel to attempt to typecast values, but to silently ignore any errors raised:

```
Album.raise_on_typecast_failure = false
album = Album.new
album.copies_sold = 'banana'
album.copies_sold # => 'banana'
```

`validates_not_string` is designed to be used in web applications where all user input comes in as strings. When the `raise_on_typecast_failure = false` setting is used in such an application, you can call `validates_not_string` with all non-string columns. If any of those columns has a string value, it's because the column was set and Sequel was not able to typecast it correctly, which means it probably isn't valid. For example, let's say that you want to check that a couple of columns contain valid dates:

```
class Album < Sequel::Model
  self.raise_on_typecast_failure = false
  def validate
    validates_not_string [:release_date, :record_date]
  end
end
album = Album.new
album.release_date = 'banana'
album.release_date # => 'banana'
album.record_date = '2010-05-17'
album.record_date # => #<Date: 4910667/2,0,2299161>
album.valid? # => false
album.errors # => {:release_date=>["is not a valid date"]}
```

For web applications, you usually want the `raise_on_typecast_failure = false` setting, so that you can accept all of the input without raising an error, and then present the user with all error messages. Without the setting, if the user submits any invalid data, Sequel will immediately raise an error. `validates_not_string` is helpful because it allows you to check for typecasting errors on non-string columns, and provides a good default error message stating that the attribute is not of the expected type.

validates_unique

`validates_unique` has a similar but different API than the other `validation_helpers` methods. It takes an arbitrary number of arguments, which should be column symbols or arrays of column symbols. If any argument is a symbol, Sequel sets up a unique validation for just that column. If any argument is an array of symbols, Sequel sets up a unique validation for the combination of the columns. This means that you get different behavior depending on whether you call the object with an array or with separate arguments. For example:

```
validates_unique(:name, :artist_id)
```

Will set up a 2 separate uniqueness validations. It will make it so that no two

albums can have the same name, and that each artist can only be associated with one album. In general, that's probably not what you want. You probably want it so that two albums can have the same name, unless they are by the same artist. To do that, you need to use an array:

```
validates_unique([:name, :artist_id])
```

That sets up a single uniqueness validation for the combination of the fields.

You can mix and match the two approaches. For example, if all albums should have a unique UPC, and no artist can have duplicate album names:

```
validates_unique(:upc, [:name, :artist_id])
```

`validates_unique` also accepts a block to scope the uniqueness constraint. For example, if you want to ensure that all active albums have a unique name, but inactive albums can duplicate the name:

```
validates_unique(:name){|ds| ds.filter(:active)}
```

If you provide a block, it is called with the dataset to use for the uniqueness check, which you can then filter to scope the uniqueness validation to a subset of the model's dataset.

Additionally, you can also include an optional options hash as the last argument. Unlike the other validations, the options hash for `validates_unique` only checks for two options:

:message	The message to use
:only_if_modified	Only check the uniqueness if the object is new or one of the columns has been modified.

`validates_unique` is the only method in `validation_helpers` that checks with the database. Attempting to validate uniqueness outside of the database suffers from a race condition, so any time you want to add a uniqueness validation, you should make sure to add a uniqueness constraint or unique index on the underlying database table. See the "Migrations and Schema Modification" guide for details on how to do that.

`validation_helpers` Options

All `validation_helpers` methods except `validates_unique` accept the following options:

:message

The most commonly used option, used to override the default validation message. Can be either a string or a proc. If a string, it is used directly. If a proc, the proc is called and should return a string. If the validation method takes an argument before the array of attributes, that argument is passed as an argument to the proc. The exception is the validates_not_string method, which doesn't take an argument, but passes the schema type symbol as the argument to the proc.

```
class Album < Sequel::Model
  def validate
    validates_presence :copies_sold, :message=>'was not given'
    validates_min_length 3, :name, :message=>proc{|s| "should be more than #{s} characters"}
  end
end
```

:allow_nil

The :allow_nil option skips the validation if the attribute value is nil or if the attribute is not present. It's commonly used when you have a validates_presence method already on the attribute, and don't want multiple validation errors for the same attribute:

```
class Album < Sequel::Model
  def validate
    validates_presence :copies_sold
    validates_integer :copies_sold, :allow_nil=>true
  end
end
```

Without the :allow_nil option to validates_integer, if the copies_sold attribute was nil, you would get two separate validation errors, instead of a single validation error.

:allow_blank

The :allow_blank is similar to the :allow_nil option, but instead of just skipping the attribute for nil values, it skips the attribute for all blank values. For example, let's say that artists can have a website. If they have one, it should be formatted like a URL, but it can be nil or an empty string if they don't have one.

```
class Album < Sequel::Model
  def validate
    validates_format /\Ahttps?:\/\//, :website, :allow_blank=>true
  end
end
```

```
a = Album.new
a.website = ''
a.valid? # true
```

`:allow_missing`

The `:allow_missing` option is different from the `:allow_nil` option, in that instead of checking if the attribute value is nil, it checks if the attribute is present in the model instance's values hash. `:allow_nil` will skip the validation when the attribute is in the values hash and has a nil value and when the attribute is not in the values hash. `:allow_missing` will only skip the validation when the attribute is not in the values hash. If the attribute is in the values hash but has a nil value, `:allow_missing` will not skip it.

The purpose of this option is to work correctly with missing columns when inserting or updating records. Sequel only sends the attributes in the values hash when doing an insert or update. If the attribute is not present in the values hash, Sequel doesn't specify it, so the database will use the table's default value when inserting the record, or not modify the value when saving it. This is different from having an attribute in the values hash with a value of nil, which Sequel will send as NULL. If your database table has a non NULL default, this may be a good option to use. You don't want to use allow_nil, because if the attribute is in values but has a value nil, Sequel will attempt to insert a NULL value into the database, instead of using the database's default.

Conditional Validation

Because Sequel uses the `validate` instance method to handle validation, making validations conditional is easy as it works exactly the same as ruby's standard conditionals. For example, if you only want to validate an attribute when creating an object:

```
validates_presence :name if new?
```

If you only want to validate the attribute when updating an existing object:

```
validates_integer :copies_sold unless new?
```

Let's say you only to make a validation conditional on the status of the object:

```
validates_presence :name if status_id > 1
validates_integer :copies_sold if status_id > 3
```

You can use all the standard ruby conditional expressions, such as `case`:

```
case status_id
```

```
when 1
  validates_presence :name
when 2
  validates_presence [:name, :artist_id]
when 3
  validates_presence [:name, :artist_id, :copies_sold]
end
```

You can make the input to some validations dependent on the values of another attribute:

```
validates_min_length(status_id > 2 ? 5 : 10, [:name])
validates_presence(status_id < 2 ? :name : [:name, :artist_id])
```

Basically, there's no special syntax you have to use for conditional validations. Just handle conditionals the way you would in other ruby code.

Default Error Messages

These are the default error messages for all of the helper methods in `validation_helpers`:

:exact_length	is not #{arg} characters
:format	is invalid
:includes	is not in range or set: #{arg.inspect}
:integer	is not a number
:length_range	is too short or too long
:max_length	is longer than #{arg} characters
:min_length	is shorter than #{arg} characters
:not_string	is not a valid #{schema_type}

:numeric	is not a number
:type	is not a #{arg}
:presence	is not present
:unique	is already taken

Modifying the Default Options

It's easy to modify the default options used by `validation_helpers`. All of the default options are stored in the `Sequel::Plugins::ValidationHelpers::DEFAULT_OPTIONS` hash. So you just need to modify that hash to change the default options. One way to do that is to use `merge!` to update the hash:

```
Sequel::Plugins::ValidationHelpers::DEFAULT_OPTIONS.merge!(
 :presence=>{:message=>'cannot be empty'},
 :includes=>{:message=>'invalid option', :allow_nil=>true},
 :max_length=>{:message=>lambda{|i| "cannot be more than #{i} characters"}, :allow_nil=>true},
 :format=>{:message=>'contains invalid characters', :allow_nil=>true})
```

This updates the default messages that will be used for the presence, includes, max_length, and format validations, and sets the default value of the `:allow_nil` option to true for the includes, max_length, and format validations.

Custom Validations

Just as the first validation example showed, you aren't limited to the validation methods defined by `validation_helpers`. Inside the `validate` method, you can add your own validations by adding to the instance's errors using `errors.add` whenever an attribute is not valid:

```
class Album < Sequel::Model
  def validate
    super
    errors.add(:release_date, 'cannot be before record date') if release_date < record_date
  end
end
```

Just like conditional validations, with custom validations you are just using the standard ruby conditionals, and calling `errors.add` with the column symbol and

the error message if you detect invalid data.

It's fairly easy to create your own custom validations that can be reused in all your models. For example, if there is a common need to validate that one column in the model comes before another column:

```
class Sequel::Model
  def validates_after(col1, col2)
    errors.add(col1, "cannot be before #{col2}") if send(col1) < send(col2)
  end
end
class Album < Sequel::Model
  def validate
    super
    validates_after(:release_date, :record_date)
  end
end
```

Setting Validations for All Models

Let's say you want to add some default validations that apply to all of your model classes. It's fairly easy to do by overriding the `validate` method in `Sequel::Model`, adding some validations to it, and if you override `validate` in your model classes, just make sure to call `super`.

```
class Sequel::Model
  def self.string_columns
    @string_columns ||= columns.reject{|c| db_schema[c][:type] != :string}
  end
  def validate
    super
    validates_format(/\A[^\x00-\x08\x0e-\x1f\x7f\x81\x8d\x8f\x90\x9d]*\z/,
     model.string_columns,
     :message=>"contains invalid characters")
  end
end
```

This will make sure that all string columns in the model are validated to make sure they don't contain any invalid characters. Just remember that if you override the `validate` method in your model classes, you need to call `super`:

```
class Album < Sequel::Model
  def validate
    super # Important!
    validates_presence :name
  end
end
```

If you forget to call `super`, the validations that you defined in `Sequel::Model` will not be enforced. It's a good idea to call super whenever you override one of

`Sequel::Model`'s methods, unless you specifically do not want the default behavior.

Sequel::Model::Errors

' As mentioned earlier, `Sequel::Model::Errors` is a subclass of Hash with a few special methods, the most common of which are described here:

add

`add` is the method used to add error messages for a given column. It takes the column symbol as the first argument and the error message as the second argument:

```
errors.add(:name, 'is not valid')
```

on

`on` is a method usually used after validation has been completed, to determine if there were any errors on a given attribute. It takes the column value, and returns an array of error messages if there were any, or nil if not:

```
errors.on(:name)
```

If you want to make some validations dependent upon the results of other validations, you may want to use `on` inside your validates method:

```
validates_integer(:release_date) if errors.on(:record_date)
```

Here, you don't care about validating the release date if there were validation errors for the record date.

full_messages

`full_messages` returns an array of error messages for the object. It's commonly called after validation to get a list of error messages to display to the user:

```
album.errors
# => {:name=>["cannot be empty"]}
album.errors.full_messages
# => ["name cannot be empty"]
```

count

`count` returns the total number of error messages in the errors.

```
album.errors.count # => 1
```

validation_class_methods

Sequel actually ships with two validation plugins. The recommended one that this guide focused on is the `validation_helpers` plugin. However, Sequel also ships with the `validation_class_methods` plugin, which uses class methods instead of instance methods to define validations. It exists mostly for legacy compatibility, but it is still supported.

Hanna RDoc template