module Sequel::Model::ClassMethods

1. lib/sequel/model/base.rb

Parent: **Model**

Class methods for Sequel::Model that implement basic model functionality.

- All of the method names in Model::DATASET_METHODS have class methods created that call the Model's dataset with the method of the same name with the given arguments.

Methods

Public Instance

1. []
2. allowed_columns
3. call
4. clear_setter_methods_cache
5. columns
6. create
7. dataset
8. dataset=
9. dataset_method_modules
10. dataset_methods
11. dataset_module
12. db
13. db=
14. db_schema
15. def_column_alias
16. def_dataset_method
17. find
18. find_or_create
19. implicit_table_name
20. include
21. inherited
22. load
23. method_added
24. no_primary_key

25. plugin
26. plugins
27. primary_key
28. primary_key_hash
29. qualified_primary_key_hash
30. raise_on_save_failure
31. raise_on_typecast_failure
32. require_modification
33. restrict_primary_key
34. restrict_primary_key?
35. restricted_columns
36. set_allowed_columns
37. set_dataset
38. set_primary_key
39. set_restricted_columns
40. setter_methods
41. simple_pk
42. simple_table
43. strict_param_setting
44. subset
45. table_name
46. typecast_empty_string_to_nil
47. typecast_on_assignment
48. unrestrict_primary_key
49. use_after_commit_rollback
50. use_transactions

Attributes

allowed_columns	[R]	Which columns should be the only columns allowed in a call to a mass assignment method (e.g. set) (default: not set, so all columns not otherwise restricted are allowed).
dataset_method_modules	[R]	Array of modules that extend this model's dataset. Stored so that if the model's dataset is changed, it will be extended with all of these modules.

dataset_methods	[R]	Hash of dataset methods with method name keys and proc values that are stored so when the dataset changes, methods defined with def_dataset_method will be applied to the new dataset.

plugins	[R]	Array of plugin modules loaded by this class
		`Sequel::Model.plugins`
		`# => [Sequel::Model, Sequel::Model::Associations]`

primary_key	[R]	The primary key for the class. Sequel can determine this automatically for many databases, but not all, so you may need to set it manually. If not determined automatically, the default is :id.
raise_on_save_failure	[RW]	Whether to raise an error instead of returning nil on a failure to save/create /save_changes/etc due to a validation failure or a before_* hook returning false.
raise_on_typecast_failure	[RW]	Whether to raise an error when unable to typecast data for a column (default: true). This should be set to false if you want to use validations to display nice error messages to the user (e.g. most web applications). You can use the validates_not_string validations (from either the validation_helpers or validation_class_methods standard plugins) in connection with option to check for typecast failures for columns that aren't blobs or strings.
require_modification	[RW]	Whether to raise an error if an UPDATE or DELETE query related to a model instance does not modify exactly 1 row. If set to

		false, Sequel will not check the number of rows modified (default: true).
restricted_columns	[R]	Which columns are specifically restricted in a call to set/update/new/etc. (default: not set). Some columns are restricted regardless of this setting, such as the primary key column and columns in Model::RESTRICTED_SETTER_METHODS.
simple_pk	[R]	Should be the literal primary key column name if this Model's table has a simple primary key, or nil if the model has a compound primary key or no primary key.
simple_table	[R]	Should be the literal table name if this Model's dataset is a simple table (no select, order, join, etc.), or nil otherwise. This and simple_pk are used for an optimization in Model.[].
strict_param_setting	[RW]	Whether new/set/update and their variants should raise an error if an invalid key is used. A key is invalid if no setter method exists for that key or the access to the setter method is restricted (e.g. due to it being a primary key field). If set to false, silently skip any key where the setter method doesn't exist or access to it is restricted.
typecast_empty_string_to_nil	[RW]	Whether to typecast the empty string (") to nil for columns that are not string or blob. In most cases the empty string would be the way to specify a NULL SQL value in string form (nil.to_s == "), and an empty string would not usually be typecast correctly for other types, so the default is true.

typecast_on_assignment	[RW]	Whether to typecast attribute values on assignment (default: true). If set to false, no typecasting is done, so it will be left up to the database to typecast the value correctly.
use_after_commit_rollback	[RW]	Whether to enable the after_commit and after_rollback hooks when saving/destroying instances. On by default, can be turned off for performance reasons or when using prepared transactions (which aren't compatible with after commit/rollback).
use_transactions	[RW]	Whether to use a transaction by default when saving/deleting records (default: true). If you are sending database queries in before_* or after_* hooks, you shouldn't change the default setting without a good reason.

Public Instance methods

[] (*args)

Returns the first record from the database matching the conditions. If a hash is given, it is used as the conditions. If another object is given, it finds the first record whose primary key(s) match the given argument(s). If no object is returned by the dataset, returns nil.

```
Artist[1] # SELECT * FROM artists WHERE id = 1
# => #<Artist {:id=>1, ...}>
Artist[:name=>'Bob'] # SELECT * FROM artists WHERE (name = 'Bob') LIMIT 1
# => #<Artist {:name=>'Bob', ...}>
```

[show source]
call (values)

Initializes a model instance as an existing record. This constructor is used by Sequel to initialize model instances when fetching records. Requires that values be a hash where all keys are symbols. It probably should not be used by

external code.

[show source]
clear_setter_methods_cache ()

Clear the setter_methods cache

[show source]
columns ()

Returns the columns in the result set in their original order. Generally, this will use the columns determined via the database schema, but in certain cases (e.g. models that are based on a joined dataset) it will use `Dataset#columns` to find the columns.

```
Artist.columns
# => [:id, :name]
```

[show source]
create (values = {}, &block)

Creates instance using new with the given values and block, and saves it.

```
Artist.create(:name=>'Bob')
# INSERT INTO artists (name) VALUES ('Bob')
Artist.create do |a|
  a.name = 'Jim'
end # INSERT INTO artists (name) VALUES ('Jim')
```

[show source]
dataset ()

Returns the dataset associated with the Model class. Raises an `Error` if there is no associated dataset for this class. In most cases, you don't need to call this directly, as Model proxies many dataset methods to the underlying dataset.

```
Artist.dataset.all # SELECT * FROM artists
```

[show source]
dataset= (ds)

Alias of set_dataset

[show source]
dataset_module (mod = nil)

Extend the dataset with a module, similar to adding a plugin with the methods defined in DatasetMethods. If a block is given, an anonymous module is created and the module_evaled, otherwise the argument should be a module.

Returns the module given or the anonymous module created.

```
Artist.dataset_module Sequel::ColumnsIntrospection
Artist.dataset_module do
  def foo
    :bar
  end
end
Artist.dataset.foo
# => :bar
Artist.foo
# => :bar
```

[show source]

db ()

Returns the database associated with the Model class. If this model doesn't have a database associated with it, assumes the superclass's database, or the first object in Sequel::DATABASES. If no Sequel::Database object has been created, raises an error.

```
Artist.db.transaction do # BEGIN
  Artist.create(:name=>'Bob')
  # INSERT INTO artists (name) VALUES ('Bob')
end # COMMIT
```

[show source]

db= (db)

Sets the database associated with the Model class. If the model has an associated dataset, sets the model's dataset to a dataset on the new database with the same options used by the current dataset. This can be used directly on Sequel::Model to set the default database to be used by subclasses, or to override the database used for specific models:

```
Sequel::Model.db = DB1
Artist.db = DB2
```

[show source]

db_schema ()

Returns the cached schema information if available or gets it from the database. This is a hash where keys are column symbols and values are hashes of information related to the column. See `Database#schema`.

```
Artist.db_schema
# {:id=>{:type=>:integer, :primary_key=>true, ...},
#  :name=>{:type=>:string, :primary_key=>false, ...}}
```

[show source]

def_column_alias (meth, column)

Create a column alias, where the column methods have one name, but the underlying storage uses a different name.

[show source]
def_dataset_method (*args, &block)

If a block is given, define a method on the dataset (if the model currently has an dataset) with the given argument name using the given block. Also define a class method on the model that calls the dataset method. Stores the method name and block so that it can be reapplied if the model's dataset changes.

If a block is not given, just define a class method on the model for each argument that calls the dataset method of the same argument name.

```
# Add new dataset method and class method that calls it
Artist.def_dataset_method(:by_name){order(:name)}
Artist.filter(:name.like('A%')).by_name
Artist.by_name.filter(:name.like('A%'))
# Just add a class method that calls an existing dataset method
Artist.def_dataset_method(:server!)
Artist.server!(:server1)
```

[show source]
find (*args, &block)

Finds a single record according to the supplied filter. You are encouraged to use Model.[] or Model.first instead of this method.

```
Artist.find(:name=>'Bob')
# SELECT * FROM artists WHERE (name = 'Bob') LIMIT 1
Artist.find{name > 'M'}
# SELECT * FROM artists WHERE (name > 'M') LIMIT 1
```

[show source]
find_or_create (cond, &block)

Like `find` but invokes create with given conditions when record does not exist. Unlike `find` in that the block used in this method is not passed to `find`, but instead is passed to `create` only if `find` does not return an object.

```
Artist.find_or_create(:name=>'Bob')
# SELECT * FROM artists WHERE (name = 'Bob') LIMIT 1
# INSERT INTO artists (name) VALUES ('Bob')
Artist.find_or_create(:name=>'Jim'){|a| a.hometown = 'Sactown'}
# SELECT * FROM artists WHERE (name = 'Jim') LIMIT 1
# INSERT INTO artists (name, hometown) VALUES ('Jim', 'Sactown')
```

[show source]

implicit_table_name ()

Returns the implicit table name for the model class, which is the demodulized, underscored, pluralized name of the class.

```
Artist.implicit_table_name # => :artists
Foo::ArtistAlias.implicit_table_name # => :artist_aliases
```

[show source]
include (mod)

Clear the setter_methods cache when a module is included, as it may contain setter methods.

[show source]
inherited (subclass)

If possible, set the dataset for the model subclass as soon as it is created. Also, make sure the inherited class instance variables are copied into the subclass.

Sequel queries the database to get schema information as soon as a model class is created:

```
class Artist < Sequel::Model # Causes schema query
end
```

[show source]
load (values)

Calls call with the values hash. Only for backwards compatibility.

[show source]
method_added (meth)

Clear the setter_methods cache when a setter method is added

[show source]
no_primary_key ()

Mark the model as not having a primary key. Not having a primary key can cause issues, among which is that you won't be able to update records.

```
Artist.primary_key # => :id
Artist.no_primary_key
Artist.primary_key # => nil
```

[show source]
plugin (plugin, *args, &blk)

Loads a plugin for use with the model class, passing optional arguments to the plugin. If the plugin is a module, load it directly. Otherwise, require the plugin from either sequel/plugins/#{plugin} or sequel_#{plugin}, and then attempt to load the module using a the camelized plugin name under Sequel::Plugins.

[show source]
primary_key_hash (value)

Returns primary key attribute hash. If using a composite primary key value such be an array with values for each primary key in the correct order. For a standard primary key, value should be an object with a compatible type for the key. If the model does not have a primary key, raises an `Error`.

```
Artist.primary_key_hash(1) # => {:id=>1}
Artist.primary_key_hash([1, 2]) # => {:id1=>1, :id2=>2}
```

[show source]
qualified_primary_key_hash (value, qualifier=table_name)

Return a hash where the keys are qualified column references. Uses the given qualifier if provided, or the table_name otherwise. This is useful if you plan to join other tables to this table and you want the column references to be qualified.

```
Artist.filter(Artist.qualified_primary_key_hash(1))
# SELECT * FROM artists WHERE (artists.id = 1)
```

[show source]
restrict_primary_key ()

Restrict the setting of the primary key(s) when using mass assignment (e.g. `set`). Because this is the default, this only make sense to use in a subclass where the parent class has used `unrestrict_primary_key`.

[show source]
restrict_primary_key? ()

Whether or not setting the primary key(s) when using mass assignment (e.g. `set`) is restricted, true by default.

[show source]
set_allowed_columns (*cols)

Set the columns to allow when using mass assignment (e.g. `set`). Using this means that any columns not listed here will not be modified. If you have any virtual setter methods (methods that end in =) that you want to be used during mass assignment, they need to be listed here as well (without the =).

It may be better to use a method such as `set_only` or `set_fields` that lets you specify the allowed fields per call.

```
Artist.set_allowed_columns(:name, :hometown)
Artist.set(:name=>'Bob', :hometown=>'Sactown') # No Error
Artist.set(:name=>'Bob', :records_sold=>30000) # Error
```

[show source]
set_dataset (ds, opts={})

Sets the dataset associated with the Model class. `ds` can be a `Symbol`, `LiteralString`, `SQL::Identifier`, `SQL::QualifiedIdentifier`, `SQL::AliasedExpression` (all specifying a table name in the current database), or a `Dataset`. If a dataset is used, the model's database is changed to the database of the given dataset. If a dataset is not used, a dataset is created from the current database with the table name given. Other arguments raise an `Error`. Returns self.

This changes the row_proc of the dataset to return model objects, extends the dataset with the dataset_method_modules, and defines methods on the dataset using the dataset_methods. It also attempts to determine the database schema for the model, based on the given dataset.

```
Artist.set_dataset(:tbl_artists)
Artist.set_dataset(DB[:artists])
```

[show source]
set_primary_key (*key)

Sets the primary key for this model. You can use either a regular or a composite primary key. To not use a primary key, set to nil or use `no_primary_key`. On most adapters, Sequel can automatically determine the primary key to use, so this method is not needed often.

```
class Person < Sequel::Model
  # regular key
  set_primary_key :person_id
end
class Tagging < Sequel::Model
  # composite key
  set_primary_key [:taggable_id, :tag_id]
end
```

[show source]
set_restricted_columns (*cols)

Set the columns to restrict when using mass assignment (e.g. `set`). Using this means that attempts to call setter methods for the columns listed here will cause an exception or be silently skipped (based on the `strict_param_setting` setting. If you have any virtual setter methods (methods that end in =) that

you want not to be used during mass assignment, they need to be listed here as well (without the =).

It's generally a bad idea to rely on a blacklist approach for security. Using a whitelist approach such as set_allowed_columns or the instance level set_only or set_fields methods is usually a better choice. So use of this method is generally a bad idea.

```
Artist.set_restricted_column(:records_sold)
Artist.set(:name=>'Bob', :hometown=>'Sactown') # No Error
Artist.set(:name=>'Bob', :records_sold=>30000) # Error
```

[show source]
setter_methods ()

Cache of setter methods to allow by default, in order to speed up new/set /update instance methods.

[show source]
subset (name, *args, &block)

Shortcut for `def_dataset_method` that is restricted to modifying the dataset's filter. Sometimes thought of as a scope, and like most dataset methods, they can be chained. For example:

```
Topic.subset(:joes, :username.like('%joe%'))
Topic.subset(:popular){num_posts > 100}
Topic.subset(:recent){created_on > Date.today - 7}
```

Allows you to do:

```
Topic.joes.recent.popular
```

to get topics with a username that includes joe that have more than 100 posts and were created less than 7 days ago.

Both the args given and the block are passed to `Dataset#filter`.

This method creates dataset methods that do not accept arguments. To create dataset methods that accept arguments, you have to use def_dataset_method.

[show source]
table_name ()

Returns name of primary table for the dataset. If the table for the dataset is aliased, returns the aliased name.

```
Artist.table_name # => :artists
Sequel::Model(:foo).table_name # => :foo
```

```
Sequel::Model(:foo___bar).table_name # => :bar
```

[show source]

unrestrict_primary_key ()

Allow the setting of the primary key(s) when using the mass assignment methods. Using this method can open up security issues, be very careful before using it.

```
Artist.set(:id=>1) # Error
Artist.unrestrict_primary_key
Artist.set(:id=>1) # No Error
```

[show source]

Hanna RDoc template