**Enumerable Intro to Active Record**

An *Enumerated Type* (also called an “enum”) is a data type with a set of named elements, each of them having a different value that can be used as constants, among other things. An enum can be implemented in various languages and frameworks, and in this article we will implement an enum in a [Ruby on Rails](http://rubyonrails.org/) application.

Rails, as you likely know, is an open source *MVC* web application framework. The M in MVC is [ActiveRecord](http://guides.rubyonrails.org/active_record_basics.html), which is an*Object Relational Mapping Framework*. ActiveRecord gives us the ability to represent models, their data, and other database operations in an object-oriented fashion. An enum is treated specially in ActiveRecord, being implemented using the [ActiveRecord::Enum](http://api.rubyonrails.org/classes/ActiveRecord/Enum.html) module.

Rails provides a very good usage of ActiveRecord::Enum, and to have a better architecture, we need to use it with the[PostgreSQL](http://www.postgresql.org/) [Enumerated Type](http://www.postgresql.org/docs/9.4/static/datatype-enum.html). Unfortunately, there is little or no mention on how to integrate these technologies.  
  
We’ll go over how to use PostgreSQL Enumerated Types with ActiveRecord::Enum.

This article is divided into 3 sections as follows:

* Introduction to ActiveRecord::Enum
* Introduction to PostgreSQL Enumerated Types
* Integrating ActiveRecord::Enum with PostgreSQL Enumerated Types

If you are already familiar with any of the above, feel free to skip it and jump into the one you like.

**Introduction to ActiveRecord::Enum**

ActiveRecord::Enum was introduced in Rails 4.1, as announced in the [release notes](http://guides.rubyonrails.org/4_1_release_notes.html#active-record-enums). It provides the ability to manage enums, mutate their values, and scope a model against any available enum value. Before you can useActiveRecord::Enum, there are 3 aspects that you need to know: database migration, declaration, and usage.

This article will use the following use case to implement ActiveRecord::Enum. We will add gender attribute to a User model, with the following possible values:

* Male
* Female
* Not sure
* Prefer not to disclose

Database Migration

ActiveRecord::Enum uses the integer data type to store the value of the enum in the database. Let’s create the database migration as follows:

bundle exec rails generate migration AddGenderToUsers gender:integer

The above code will generate a new migration:

# db/migrate/20150619131527\_add\_gender\_to\_users.rb

class AddGenderToUsers < ActiveRecord::Migration

def change

add\_column :users, :gender, :integer

end

end

We can add a default value to the migration. And to increase application performance, we can also add an index to the column by adding the following code:

# db/migrate/20150619131527\_add\_gender\_to\_users.rb

class AddGenderToUsers < ActiveRecord::Migration

def change

# add `default: 3`, and `index: true`

add\_column :users, :gender, :integer, default: 0, index: true

end

end

Finally, run the migration:

bundle exec rake db:migrate

We are done with the database migration. Next, we need to declare the ActiveRecord::Enum to the model.

**Declaration**

There are 2 kinds of declaration for ActiveRecord::Enum. Using our use case, we will add gender to the User model. The first declaration is using *Array*:

# app/models/user.rb

class User < ActiveRecord::Base

enum gender: [ :male, :female, :not\_sure, :prefer\_not\_to\_disclose ]

end

The above code will store male value to 0, and the rest of the values mapped with their order in the array. Future additions must be added to the end of the array, and reordering or removing values must not be done. In order to be able to manipulate values later, use the second type of declaration using Hash:

# app/models/user.rb

class User < ActiveRecord::Base

enum gender: { male: 0, female: 1, not\_sure: 2, prefer\_not\_to\_disclose: 3 }

end

With this declaration, we can reorder and remove any value. The value can start from any number, as long as it is aligned with the default value in the database migration. Future additions can be put in any position.

**Usage**

There are 3 aspects of ActiveRecord::Enum usage, which are: mutation, retrieval, and scoping. Helper methods were provided to do those 3 things. With our use case, we can mutate the user’s gender as follows:

# mutate enum using the exclamation mark (!) method.

user.male!

user.female!

user.not\_sure!

user.prefer\_not\_to\_disclose!

# or mutate enum using value assignment

user.gender = nil

user.gender = "male"

user.gender = "female"

user.gender = "not\_sure"

user.gender = "prefer\_not\_to\_disclose"

To retrieve the enum value:

# retrieve enum value using the question mark (?) method

user.male? # => false

user.female? # => false

user.not\_sure? # => false

user.prefer\_not\_to\_disclose? # => true

# or retrieve using the enum name

user.gender # => "prefer\_not\_to\_disclose"

To scope the User model to the enum values, use the following methods:

# scope using enum values

User.male

User.female

User.not\_sure

User.prefer\_not\_to\_disclose

# or we can scope it manually using query with provided class methods

User.where("gender <> ?", User.genders[:prefer\_not\_to\_disclose])

As previously mentioned, ActiveRecord::Enum uses integers to stores the value in the database. Unfortunately, the enum value in the database becomes less meaningful and is not self-explanatory. Enum values becomes dependent on a model, becoming highly coupled with that model. To uncouple the enum value from a model, we need to store it using an enumerated type in the database.

**Introduction to PostgreSQL Enumerated Types**

PostgreSQL provides Enumerated Type to store a static and ordered set of values. Let’s take a look at how this is implemented in PostgreSQL. With our use case, we can create a gender type as follows:

CREATE TYPE gender AS ENUM ('male', 'female', 'not\_sure', 'prefer\_not\_to\_disclose');

Once created, we can use the type in our table like any other type. As an example, we can create users table with a gender type attribute as follows.

CREATE TABLE users (

name text,

gender gender

);

By using an enum instead of an integer, we gain type safety. That means you cannot add a value that is not part of the enum values. Here is an example of the type safety feature:

INSERT INTO users(name, gender) VALUES ('John Doe', 'male');

INSERT INTO users(name, gender) VALUES ('Confused John Doe', 'not\_sure');

INSERT INTO users(name, gender) VALUES ('Unknown John Doe', 'unknown');

ERROR: invalid input value for enum gender: "unknown"

An enumerated type is case sensitive, that means male is different from MALE. White space is also significant.

**Integrating ActiveRecord::Enum with PostgreSQL Enumerated Types**

There are 2 things that we need to do before we can use ActiveRecord::Enum with PostgreSQL Enumerated Types: database migration and enum declaration.

First, let’s create the database migration:

bundle exec rails generate migration AddGenderToUsers gender:gender

Next, edit the generated migration to add the type:

# db/migrate/20150619131527\_add\_gender\_to\_users.rb

class AddGenderToUsers < ActiveRecord::Migration

def up

execute <<-SQL

CREATE TYPE gender AS ENUM ('male', 'female', 'not\_sure', 'prefer\_not\_to\_disclose');

SQL

add\_column :users, :gender, :gender, index: true

end

def down

remove\_column :users, :gender

execute <<-SQL

DROP TYPE gender;

SQL

end

end

Once you’re finished with that, run the migration:

bundle exec rake db:migrate

Now, we have completed the database migration. The next step is to declare an enum in the User model. Earlier, we used both the Array and Hash forms to declare an enum. For the integration to work, we need to declare an enum using the Hash form:

# app/models/user.rb

class User < ActiveRecord::Base

enum gender: {

male: 'male',

female: 'female',

not\_sure: 'not\_sure',

prefer\_not\_to\_disclose: 'prefer\_not\_to\_disclose'

}

end

Finally, we can store ActiveRecord::Enum values using PostgreSQL Enumerated Types. As a bonus, all helper methods provided by ActiveRecord::Enum still work as expected.

**Conclusion**

ActiveRecord::Enum, by default, stores values using integers. The value is meaningless without the model, and it is not self-explanatory. In other words, the enum value is highly coupled with a model. To have a better architecture, we need to uncouple the enum value from a model.

PostgreSQL Enumerated Types provide a good complement for ActiveRecord::Enum. We can have a meaningful, and self-explanatory data that doesn’t require a model to decipher the value. With the type safety feature provided by PostgreSQL, we can have a solid foundation for storing ActiveRecord::Enum values.