16 APRIL 2018 / #RUBY

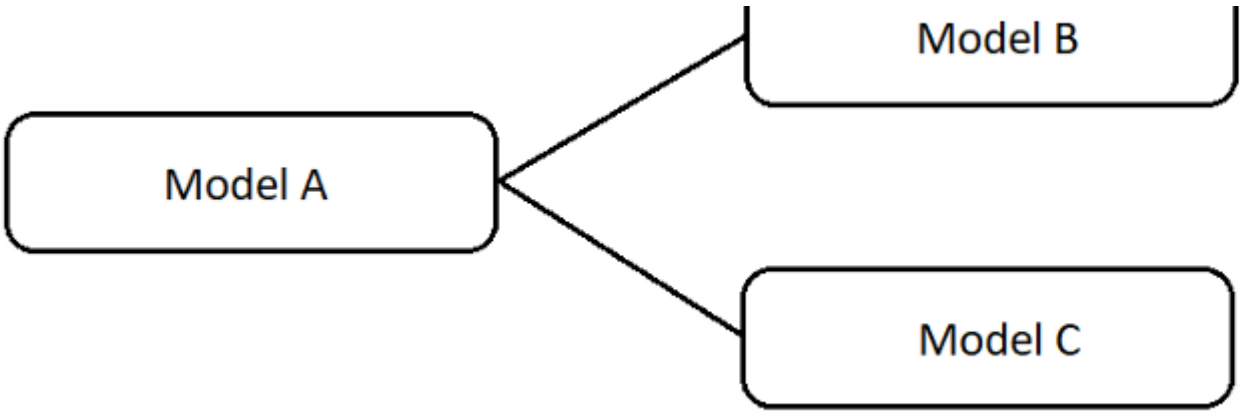
Single-table inheritance vs. polymorphic associations in Rails: find what works for you

by Haley Mnatzaganian

If you've ever created an application with more than one model, you've had to think about what type of relationships to use between those models.

As an application's complexity grows, it can be difficult to decide which relationships should exist between your models.

A situation that frequently comes up is when several of your models need to have access to the functionality of a third model. Two methods that Rails gives us to deal with this event are **single-table inheritance** and **polymorphic association.**



In Single-Table Inheritance (STI), many subclasses inherit from one superclass with all the data in the same table in the database. The superclass has a "type" column to determine which subclass an object belongs to.

In a **polymorphic association**, one model "belongs to" several other models using a single association. Each model, including the polymorphic model, has its own table in the database.

Let's take a look at each method to see when we would use them.

Single-Table Inheritance

A great way to know when STI is appropriate is when your models have **shared data/state**. Shared behavior is optional.

Let's pretend we are creating an app that lists different vehicles that are for sale at a local dealership. This dealership sells cars, motorcycles, and bicycles.

(I know dealerships don't sell bicycles, but bear with me for a minute — you'll see where I'm going with this.)

For each vehicle, the dealership wants to track the price, color, and whether the vehicle was purchased. This situation is a perfect candidate for STI, because we are using the same data for each class.

We can create a superclass `Vehicle` with the attributes for color, price, and purchased. Each of our subclasses can inherit from `Vehicle` and can all get those same attributes in one fell swoop.

Our migration to create the vehicles table might look like this:

```
class CreateVehicles < ActiveRecord::Migration[5.1]  def change                                c
```

It is important that we create the `type` column for the superclass. This tells Rails that we are using STI and want all the data for `Vehicle` and its subclasses to be in the same table in the database.

Our model classes would look like this:

```
class Bicycle < Vehicleend
```

```
class Motorcycle < Vehicleend
```

```
class Car < Vehicleend
```

This setup is great because any methods or validations in the `Vehicle` class are shared with each of its subclasses. We can add unique methods to any of the subclasses as needed. They are independent of each other and their behavior is not shared horizontally.

Additionally, since we know that the subclasses share the same data fields, we can make the same calls on objects from different classes:

```
mustang = Car.new(price: 50000, color: red)harley = Motorcycle.new(price: 30000, color: black)
```

```
mustang.price=> 50000
```

```
harley.price=> 30000
```

Umm, where can I find this dealership? (source)

Adding functionality

Now let's say the dealer decides to collect some more information about the vehicles.

For `Bicycles`, she wants to know if each bike is a road, mountain, or hybrid bike. And for `Cars` and `Motorcycles`, she wants to keep track of the horsepower.

So we create a migration to add `bicycle_type` and `horsepower` to the `Vehicles` table.

All of a sudden, our models don't perfectly share data fields anymore. Any `Bicycle` object will not have a `horsepower` attribute, and any `Car` or `Motorcycle` will not have a `bicycle_type` (hopefully — I'll get to this in a moment).

Yet every bicycle in our table will have a `horsepower` field, and every car and motorcycle will have a `bicycle_type` field.

This is where things can get sticky. A few issues can arise in this situation:

1. Our table will have a lot of null values (`nil` in Ruby's case) since objects will have fields that don't apply to them. These `nulls` can cause problems as we add validations to our models.
2. As the table grows, we can run into performance costs when querying if we don't add filters. A search for a certain `bicycle_type` will look at **every item** in the table— so not only `Bicycles`, but `Cars` and `Motorcycles` also.
3. As is, there is nothing stopping a user from adding "inappropriate" data to the wrong model. For example, a user with some know-how could create a `Bicycle` with a `horsepower` of 100. We would need validations and good app design to prevent the creation of an invalid object.

So, as we can see, STI does have some flaws. It is great for applications where your models share data fields and aren't likely to change.

STI PROS:

- Simple to implement
- DRY — saves replicated code using inheritance and shared attributes
- Allows subclasses to have own behavior as necessary

STI CONS:

- Doesn't scale well: as data grows, table can become large and possibly difficult to maintain/query
- Requires care when adding new models or model fields that deviate from the shared fields
- (conditional) Allows creation of invalid objects if validations are not in place
- (conditional) Can be difficult to validate or query if many null values exist in table

With polymorphic associations, a model can `belong_to` several models with a single association.

It's morphin' time. (source)

This is useful when several models do not have a relationship or share data with each other, but have a relationship with the polymorphic class.

As an example, let's think of a social media site like Facebook. On Facebook, both individuals and groups can share posts.

The individuals and groups are not related (other than both being a type of user), and so they have different data. A group probably has fields like `member_count` and `group_type` that don't apply to an individual, and vice-versa).

Without polymorphic associations, we would have something like this:

```
class Post  belongs_to :person  belongs to :groupend



class Person  has_many :postsend



class Group  has_many :postsend
```

Normally, to find out who owns a certain profile, we look at the column that is the `foreign_key`. A

However, our Posts table would have two competing foreign keys: `group_id` and `person_id`. This would be problematic.

When trying to find the owner of a post, we would have to make a point to check both columns to find the correct foreign_key, rather than relying on one. What happens if we run into a situation where both columns have a value?

A polymorphic association addresses this issue by condensing this functionality into one association. We can represent our classes like this:

```
class Post  belongs_to :postable, polymorphic: trueend
```

```
class Person  has_many :posts, as :postableend
```

```
class Group  has_many :posts, as :postableend
```

The Rails convention for naming a polymorphic association uses “-able” with the class name (`:postable` for the `Post` class). This makes it clear in your relationships which class is polymorphic. But you can use whatever name for your polymorphic association that you like.

To tell our database we’re using a polymorphic association, we use special “type” and “id” columns for the polymorphic class.

The `postable_type` column records which model the post belongs to, while the `postable_id` column tracks the id of the owning object:

```
haley = Person.first=> returns Person object with name: "Haley"
```

```
article = haley.posts.firstarticle.postable_type=> "Person"
```

```
article.postable_id=> 1 # The object that owns this has an id of 1 (in this case a      Person)
```

```
new_post = haley.posts.new()# Automatically fills in postable_type and postable_id using haley
```

of this, you can act the same way you would when using two models that have a belongs_to association.

Note: polymorphic associations work with both has_one and has_many associations.

```
haley.posts# returns ActiveRecord array of posts
```

```
haley.posts.first.content=> "The content from my first post was a string..."
```

One difference is going “backwards” from a post to access its owner, since its owner could come from one of several classes.

To do that quickly, you need to add a foreign key column and a type column to the polymorphic class. You can find the owner of a post using `postable`:

```
new_post.postable=> returns Person object
```

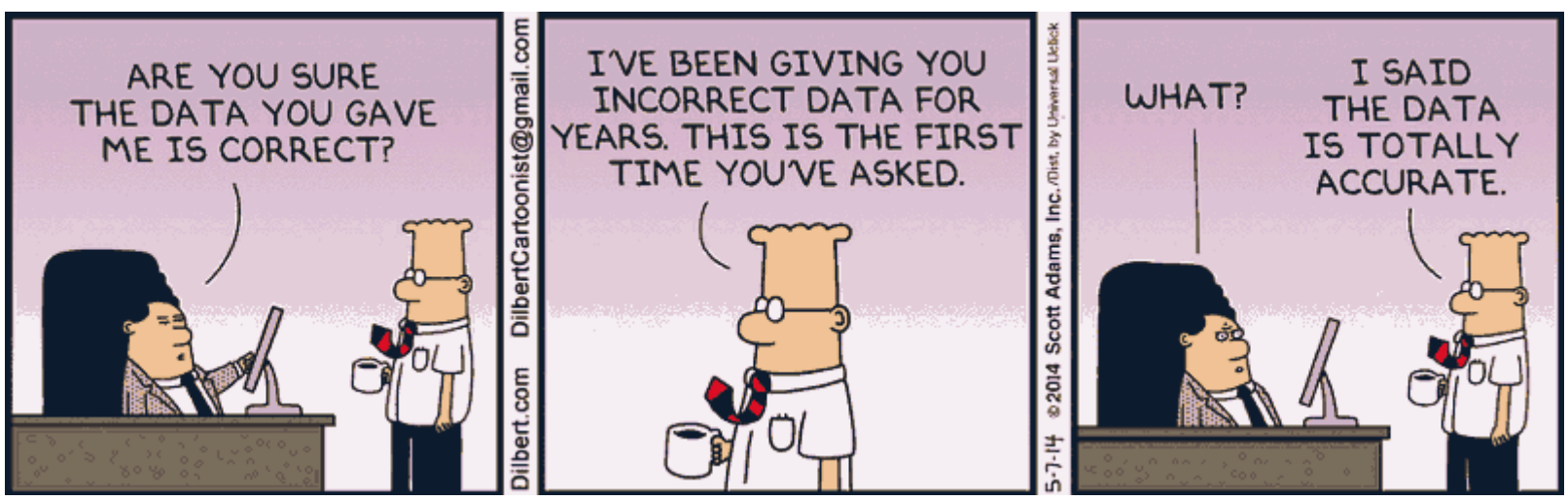
```
new_post.postable.name=> "Haley"
```

Additionally, Rails implements some security within polymorphic relationships. Only classes that are part of the relationship can be included as a `postable_type`:

```
new_post.update(postable_type: "FakeClass")=> NameError: uninitialized constant FakeClass
```

Warning

Polymorphic associations come with one huge red flag: **compromised data integrity.**



In a normal belongs_to relationship, we use foreign keys for reference in an association.

They have more power than just forming a link, though. Foreign keys also prevent referential errors by requiring that the object referenced in the foreign table does, in fact, exist.

If someone tries to create an object with a foreign key that references a null object, they will get an error.

Unfortunately, polymorphic classes can't have foreign keys for the reasons we discussed. We use the `type` and `id` columns in place of a foreign key. This means we lose the protection that foreign keys offer.

Rails and ActiveRecord help us out on the surface, but anyone with direct access to the database can create or update objects that reference null objects.

For example, check out this SQL command where a post is created even though the group it is associated with doesn't exist.

```
Group.find(1000)=> ActiveRecord::RecordNotFound: Couldn't find Group with 'id'=1000



# SQLINSERT INTO POSTS (postable_type, postable_id) VALUES ('Group', 1000)=> # returns success
```

Thankfully, proper application setup can prevent this from being possible. Because this is a serious issue, you should only use polymorphic associations when your database is contained. If other applications or databases need to access it, you should consider other methods.

Polymorphic association PROS:

- Easy to scale in amount of data: information is distributed across several database tables to minimize table bloat
- Easy to scale number of models: more models can be easily associated with the polymorphic class
- DRY: creates one class that can be used by many other classes

Polymorphic association CONS

- More tables can make querying more difficult and expensive as the data grows. (Finding all posts that were created in a certain time frame would need to scan all associated tables)
- Cannot have foreign key. The `id` column can reference any of the associated model tables, which can slow down querying. It must work in conjunction with the `type` column.
- If your tables are very large, a lot of space is used to store the string values for `postable_type`

How to know which method to use

STI and polymorphic associations have some overlap when it comes to use cases. While not the only solutions to a “tree-like” model relationship, they both have some obvious advantages.

Both the `Vehicle` and `Postable` examples could have been implemented using either method. However, there were a few reasons that made it clear which method was best in each situation.

Here are four factors to consider when deciding whether either of these methods fits your needs.

1. **Database structure.** STI uses only one table for all classes in the relationship, while polymorphic associations use a table per class. Each method has its own advantages and disadvantages as the application grows.
2. **Shared data or state.** STI is a great option if your models have many shared attributes. Otherwise a polymorphic association is probably the better choice.
3. **Future concerns.** Consider how your application might change and grow. If you’re considering STI but think you’ll add models or model fields that deviate from the shared structure, you might want to rethink your plan. If you think your structure is likely to remain the same, STI will *generally* be faster for querying.
4. **Data integrity.** If data is not going to be contained (one application using your database), polymorphic association is probably a bad choice because your data will be compromised.

Final Thoughts

Neither STI nor polymorphic associations are perfect. They both have pros and cons that often make one or the other more fit for associations with many models.

I wrote this article to teach myself these concepts just as much as to teach them to anyone else. If there is anything incorrect or any points you think should be mentioned, please help me and everyone else out by sharing in the comments!

If you learned something or found this helpful, please click on the ? button to show your support!

You can make a tax-deductible donation here.

Our Nonprofit

About

Alumni Network

Open Source

Shop

Support

Sponsors

Academic Honesty

Code of Conduct

Privacy Policy

Terms of Service

Copyright Policy

Trending Guides

2019 Web Developer Roadmap

Python Tutorial

CSS Flexbox Guide

JavaScript Tutorial

Python Example

HTML Tutorial

Linux Command Line Guide

JavaScript Example

Git Tutorial

React Tutorial

Java Tutorial

Linux Tutorial

CSS Tutorial

jQuery Example

SQL Tutorial

CSS Example

React Example

Angular Tutorial

Bootstrap Example

How to Set Up SSH Keys

WordPress Tutorial

PHP Example