Advanced Migrations

Let's quickly generate a new app to play with for the next few lectures. It's an app to manage a coding school.

```
$ cd YOUR_PROJECTS_FOLDER
$ rails new good_schooler
$ cd good_schooler
```

The app should have only one model - the most common model in web development: the ubiquitous user class.

```
$ rails g scaffold user name
$ rake db:migrate
```

(You may also want go get this app running on PostgreSQL instead of SQLite, but it's not required. I've tried to provide output for both databases in this tutorial.)

Our `User` model is misleadingly simple.

app/models/user.rb

```
class User < ActiveRecord::Base
end
```

That's because the attribute we requested (a `name` string) was defined in a migration file.

db/migrate/20140605210634_create_users.rb

```
class CreateUsers < ActiveRecord::Migration
  def change
    create_table :users do |t|
      t.string :name

      t.timestamps
    end
  end
end
```

Why is the name attribute defined in the migration instead of in the model (i.e. with `attr_accessor`)?

Because then the information would be repeated. It would be defined once within the database and once in our Ruby code. And that comes with all the associated problems with "wet" code, in

particular synchronization. What if the Ruby code says my model has a `first_name` and `last_name` column, but the database only has a `name` column?

So Rails prefers to define the fields once, in the database. And our seemingly empty ActiveRecord model will just pull the attribute names from there. (Other web frameworks repeat the attributes in both places - I guess they're just not as smart as Rails.)

Adding & Removing Columns

If you screw up generating a scaffold, you can always undo your work with the Rails destroyer ("`rails destroy`" or "`rails d`").

But that's a bit harsh. I don't want to delete and re-create my app every time I decide to add a new field to my user model. What if I made changes to the code? What if I added validations?

To edit your model after you've generated the scaffold, you can use a migration.

```
$ rails g migration
Usage:
  rails generate migration NAME [field[:type][:index] field[:type][:index]] [options]
```

When you want to create a new model, Rails can generate a model for you. When you want to *edit* an existing model, Rails can generate a migration.

The documentation for generating a migration says I need to specify a `NAME` (required) followed by a list of `field`s (optional) and `options` (optional). So let's try that.

```
$ rails g migration AddEmail
    invoke  active_record
    create    db/migrate/20140606183930_add_email.rb
```

db/migrate/20140606183930_add_email.rb

```
class AddEmail < ActiveRecord::Migration
  def change
  end
end
```

That wasn't very exciting. Rails generated an empty migration called `AddEmail`. We can do better. Let's delete this migration and try again.

```
$ rails d migration AddEmail
    invoke  active_record
    remove    db/migrate/20140606183930_add_email.rb
```

```
$ rails g migration AddEmailToUser email
    invoke  active_record
    create    db/migrate/20140606184230_add_email_to_user.rb
```

db/migrate/20140606184230_add_email_to_user.rb

```
class AddEmailToUser < ActiveRecord::Migration
  def change
    add_column :users, :email, :string
  end
end
```

Now that's interesting. Rails created a migration to add a string `email` column to my `users` table. How did it figure that out?

Migrations, like much of Rails, are magic. But it's not a black magic/voodoo. It's actually easy to understand.

Any migration name of the form AddXXXToYYY, followed by a list of fields, will automatically generate a migration to add the listed fields to the specified table named YYY:

```
rails g migration AddEmailToUser email
```

From this command, Rails can determine that the model you want to update is called `User` (and, by convention, the table is `users`) and the way you want to update it is to *add* a field called `email`. Similarly:

```
$ rails g migration RemoveEmailFromUser email
    invoke  active_record
    create    db/migrate/20140606184805_remove_email_from_user.rb
```

db/migrate/20140606184805_remove_email_from_user.rb

```
class RemoveEmailFromUser < ActiveRecord::Migration
  def change
    remove_column :users, :email, :string
  end
end
```

Here, Rails can determine that we want to create a migration to *remove* the `email` column from the `users` table.

By default we've been adding (and removing) string attributes. If we want a number instead, we can specify a database data type. Just follow the attribute name with a `:` and a *type*.

```
$ rails g migration AddAgeToUser age:integer
    invoke  active_record
    create    db/migrate/20140606185113_add_age_to_user.rb
```

db/migrate/20140606185113_add_age_to_user.rb

```
class AddAgeToUser < ActiveRecord::Migration
  def change
    add_column :users, :age, :integer
  end
end
```

The supported types are:

- integer
- primary_key
- decimal
- float
- boolean
- binary
- string
- text
- date
- time
- datetime
- timestamp

These are not Ruby data types. These are database data types (or, perhaps more accurately, SQL language data types)

Ruby variables (and Ruby's object attributes) aren't *strongly-typed*. If I assign a Ruby variable to a number, I can easily reassign it to a string without a problem.

irb

```
a = 1
a = "hello world"
a = {}
a = []
```

But database columns *are* strongly-typed. If I say a database column should contain a number, I should only place numbers in that column. Otherwise "bad things" will happen (depending on the type of database you're using).

So let's migrate.

```
$ rake db:migrate
== 20140606184230 AddEmailToUser: migrating ===================================
-- add_column(:users, :email, :string)
   -> 0.0491s
== 20140606184230 AddEmailToUser: migrated (0.0493s) ==========================

== 20140606184805 RemoveEmailFromUser: migrating ==============================
-- remove_column(:users, :email, :string)
   -> 0.0238s
== 20140606184805 RemoveEmailFromUser: migrated (0.0239s) =====================

== 20140606185113 AddAgeToUser: migrating =====================================
-- add_column(:users, :age, :integer)
   -> 0.0010s
== 20140606185113 AddAgeToUser: migrated (0.0011s) ============================
```

Oops! I didn't mean to do *all* of these migrations. These were just demonstrations. People are going to think I'm crazy when they look at Git and see that I added a field, then deleted the same field again. I need to undo these migrations.

Let's talk about fixing migration errors.

Undoing Migrations

There's a command for undoing bad migrations.

```
$ rake db:rollback
== 20140606185113 AddAgeToUser: reverting =====================================
-- remove_column(:users, :age, :integer)
   -> 0.0019s
== 20140606185113 AddAgeToUser: reverted (0.0401s) ============================
```

The `db:rollback` task will rollback my database one migration. There's some magic going on here. In my last migration, I generated code to *add* an (integer) `age` column. Rails figured out on it's own that, to undo that migration, I need to *remove* the `age` column. Smart, that Rails.

I can rollback 2 more times to get back to where I started. Or I can take the express train and migrate exactly to where I want to be.

```
$ rake db:migrate VERSION=20140605210634_create_users
== 20140606184805 RemoveEmailFromUser: reverting ==============================
-- add_column(:users, :email, :string)
```

```
  -> 0.0012s
== 20140606184805 RemoveEmailFromUser: reverted (0.0040s) ======================

== 20140606184230 AddEmailToUser: reverting ====================================
-- remove_column(:users, :email, :string)
  -> 0.0009s
== 201406061r84230 AddEmailToUser: reverted (0.0010s) ==========================
```

By specifying a `VERSION`, I was able to use a single command to skip exactly to the database version I want, right after I created the users table.

Now I can delete the bad migrations. I can `rm` the files in db/migrate myself or I can ask Rails to destroy them for me. Destroying sounds like more fun, so let's do that.

```
$ rails d migration AddAgeToUser
    invoke  active_record
    remove    db/migrate/20140606185113_add_age_to_user.rb
$ rails d migration RemoveEmailFromUser
    invoke  active_record
    remove    db/migrate/20140606184805_remove_email_from_user.rb
```

In general, migrations shouldn't be edited or deleted. They're an audit trail of all the changes you've made to your database, and it would be very confusing if that trail were altered to make it nonsensical or full of gaps.

But if you haven't pushed your changes to GitHub yet, you still have time to change your models before anyone else finds out. Just make sure you change them *after* you've rolled them back. Otherwise, Rails is going to have a hard time figuring out how to undo your mistake.

When you make a migration mistake.

1. **Rollback** the migration *first* (`rake db:rollback`)
2. **Fix** the mistake (*edit* or *delete* the migration)
3. **Migrate** again (`rake db:migrate`)

But what if I really really screwed up?

In a worst case scenario, you have few options. If you don't care about your data, you can start over with:

```
rake db:drop db:create db:migrate
```

All your data will be deleted, but sometimes a full reset is the only way to move forward.

If you'd like to keep some of your data, or if the full reset failed, then come ask for help.

Confirming the Migration

Now I'm lost. What db version am I at now?

```
$ rake db:version
Current version: 20140605210634
```

This corresponds to the timestamp of the migration last ran.

```
$ ls db/migrate/
20140605210634_create_users.rb          20140606184230_add_email_to_user.rb
```

Ok, so I'm at the `create_users` migration. I'm now ready to add the `email` field (again).

```
$ rake db:migrate
== 20140606184230 AddEmailToUser: migrating ===================================
-- add_column(:users, :email, :string)
   -> 0.0011s
== 20140606184230 AddEmailToUser: migrated (0.0012s) ==========================
```

Looks good. Let's see if it works.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> u = User.new
irb(main):002:0> u.email = 'ed@wyncode.co'
=> "ed@wyncode.co"
irb(main):003:0> u.name = 'Ed'
=> "ed"
irb(main):004:0> u.save
=> true
irb(main):005:0> exit
```

(can also just type `User.connection` followed by `User`)

Let's use SQL to confirm that the database looks the way we think it should. It turns out that Rails has anticipated our need to occasionally peer into the database:

```
$ rails db
```

That's `rails db`, not `rake db`.

rails db (psql)

```
wyncode=# select * from users;
 id | name |         created_at         |         updated_at         |     email
----+------+----------------------------+----------------------------+---------------
  1 | Ed   | 2014-06-06 19:22:47.099676 | 2014-06-06 19:23:26.399655 | ed@wyncode.co
(1 row)
wyncode=# \q
```

rails db (sqlite)

```
sqlite> .mode column
sqlite> .headers on
sqlite> select * from users;
id          name        created_at                  updated_at                  email
----------  ----------  --------------------------  --------------------------  ------------
1                       2016-02-05 21:17:23.409605  2016-02-05 21:17:23.409605  ed@wyncode.co
sqlite> .exit
```

Looks right. The column was added, and I was able to populate it with an email address.

Unique Migration

I want the email address to be unique. I know how to do that with an ActiveRecord validation, but I want something stronger.

Databases also have validators. And Rails knows how to use them.

```
$ rails g migration AddEmailUniqToUser email:uniq
    invoke  active_record
    create    db/migrate/20140606193440_add_email_uniq_to_user.rb
```

db/migrate/20140606193440_add_email_uniq_to_user.rb

```
class AddEmailUniqToUser < ActiveRecord::Migration
  def change
    add_column :users, :email, :string
    add_index :users, :email, unique: true
  end
end
```

That is close to what I want. I don't need to add the email column again. I already have one. But I do want the index. So I'll delete that `add_column` line and keep the `add_index` line.

db/migrate/20140606193440_add_email_uniq_to_user.rb

```
class AddEmailUniqToUser < ActiveRecord::Migration
```

```
  def change
    #add_column :users, :email, :string
    add_index :users, :email, unique: true
  end
end
```

```
$ rake db:migrate
== 20140606193440 AddEmailUniqToUser: migrating ===============================
-- add_index(:users, :email, {:unique=>true})
  -> 0.0433s
== 20140606193440 AddEmailUniqToUser: migrated (0.0436s) ======================
```

Let's confirm that it's working by trying to add a user with a duplicate email.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> User.create(name: 'Ed2', email: 'ed@wyncode.co')
```

psql error

```
ActiveRecord::RecordNotUnique: PG::UniqueViolation: ERROR:  duplicate key value violates
unique constraint "index_users_on_email"
DETAIL:  Key (email)=(ed@wyncode.co) already exists.
```

sqlite error

```
ActiveRecord::RecordNotUnique: SQLite3::ConstraintException: UNIQUE constraint failed:
users.email: INSERT INTO "users" ("name", "email", "created_at", "updated_at") VALUES (?, ?,
?, ?)
```

The database is complaining about duplicate email addresses. Mission accomplished.

But why? Why do this when we could use ActiveRecord instead?

First, it may be faster (and it may not be). We'll talk about database "indexing" in a few.

Second, other people besides Rails can use databases. When adding a validation to Rails, anybody that uses your app won't be able to do bad things. But that won't stop someone from sending bad SQL to your database from outside of Rails. These database-level validations are a last line of defense against bad data.

Index Migration

When we asked the database to enforce email uniqueness, is used `add_index` with `unique: true`. But you can also add a non-unique index.

Why would you do that?

Because it makes your database faster. And, as we'll learn later, a big reason why a lot of web apps are slow is because their databases are slow.

Say we have an app that frequently looks up users by name. These lines of code runs frequently in our app:

```
User.find_by(name: SOMETHING)
User.where(name: SOMETHING)
```

If we'd like to make that code faster, we can add a database index.

```
$ rails g migration AddNameIndexToUser name:index
    invoke  active_record
    create    db/migrate/20140606202028_add_name_index_to_user.rb
```

db/migrate/20140606202028_add_name_index_to_user.rb

```
class AddNameIndexToUser < ActiveRecord::Migration
  def change
    add_column :users, :name, :string
    add_index :users, :name
  end
end
```

Again, I don't need the `add_column`. I only need the `add_index`. So let's edit and migrate.

db/migrate/20140606202028_add_name_index_to_user.rb

```
class AddNameIndexToUser < ActiveRecord::Migration
  def change
    add_index :users, :name
  end
end
```

```
$ rake db:migrate
== 20140606202028 AddNameIndexToUser: migrating ===============================
-- add_index(:users, :name)
  -> 0.0276s
== 20140606202028 AddNameIndexToUser: migrated (0.0278s) ======================
```

How do I know my app is faster?

Good question! A lot of people do things they think will make their apps faster without actually checking if they made a difference.

To get some confirmation that this index is working, we need to create some more Users.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> User.create(name: 'Jo')
...
irb(main):002:0> User.create(name: 'Juha')
...
```

Now that we have more than one user, let's see if we can get the database to *explain* what's going on.

```
$ rails c
irb(main):001:0> User.where(name: '').explain
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE "users"."name" = ?  [["name", ""]]
=> EXPLAIN for: SELECT "users".* FROM "users" WHERE "users"."name" = ? [["name", ""]]
0|0|0|SEARCH TABLE users USING INDEX index_users_on_name (name=?)
```

So, when we query for all the users with some name (it doesn't matter what), the database searches *using* an *index*.

Let's undo the migration and see what it looked like before.

```
$ rake db:rollback
$ rails c
irb(main):001:0> User.where(name: '').explain
  User Load (0.2ms)  SELECT "users".* FROM "users" WHERE "users"."name" = ?  [["name", ""]]
=> EXPLAIN for: SELECT "users".* FROM "users" WHERE "users"."name" = ? [["name", ""]]
0|0|0|SCAN TABLE users
```

Without the index in place, the database must *scan* the users table, meaning it starts at row 1 and examines every row until it finds what it's looking for.

What's the big-O notation of that?

It's O(n).

With the index, database search performance boosts to *approx*. O(1).

What does O(1) mean?

So we want this index. Let's undo the rollback.

```
$ rake db:migrate
```

In fact, we should index every column, all the time, so all of my ActiveRecord code is this fast.

Not so fast. This performance comes at a price. *Reading* data is faster, but *writing* data is slower. Data must be written twice, once to the table and again to the index. (How an index works should have been covered in the homework.)

Scaling a *read-heavy* site is easy. Just add indices.

Scaling a *write-heavy* site is hard, no matter what programming language or framework you use.

Not-Null Migration

I don't always need Rails to generate the migration code for me. I can write one myself.

But I should still use Rails to, at the very least, generate a blank one for me. Because that timestamp in the beginning is important. That's how Rails keeps the migrations in order. And it's annoying to come up with a timestamp on my own.

Say I want the database to make sure my email addresses aren't `null` (SQL's `null` is Ruby's `nil`). Sure, I can use an ActiveRecord validation for this, but that only protects Rails database clients, not every client.

Unfortunately, there isn't a Rails shortcut for creating a *non-null* database restriction. There's only :`uniq` and :`index`. So let's generate a blank one ...

```
$ rails g migration ChangeEmailNotNullInUser
    invoke  active_record
    create    db/migrate/20160206002740_change_email_not_null_in_user.rb
```

… and fill it in.

db/migrate/20160206002740_change_email_not_null_in_user.rb

```
class ChangeEmailNotNullInUser < ActiveRecord::Migration
```

```
  def change
    change_column :users, :email, :string, null: false
  end
end
```

Instead of adding or removing files or indices, we're *changing an existing column*.

In this example, we're changing a column to add a new restriction.

You can also change a column's type. So if you made a column a "string", but you want it to be an "integer" instead, you could also change it with `change_column`.

Let's run this migration.

```
$ rake db:migrate
== 20160206002740 ChangeEmailNotNullInUser: migrating ========================
-- change_column(:users, :email, :string, {:null=>false})
rake aborted!
StandardError: An error has occurred, this and all later migrations canceled:

PostgreSQL error
PG::NotNullViolation: ERROR:  column "email" contains null values
SQLite error
SQLite3::ConstraintException: NOT NULL constraint failed: users.email
```

There is a problem. I want the database to prevent `null` values in the `email` column, but there are *already* `null` values there. I can't prevent bad data from getting into my database in the future until I figure out how to clean out the bad data that's there already.

This is why bad data is such a pain. Once you discover it, it's hard to fix.

db/migrate/20140609051816_add_email_not_null_to_user.rb

```
class AddEmailNotNullToUser < ActiveRecord::Migration
  def change
    # set a default for users with no email
    User.where(email: nil).each do |user|
      user.email = "#{user.name}@wyncode.co"
      user.save
    end
    change_column :users, :email, :string, null: false
  end
end
```

I've updated my migration to set a default email address for each `User` that doesn't have one. I've used the `where` finder method to retrieve a list of all users with no email address, then I set a default email for each, one by one. It's not the best solution to this problem (these emails probably don't work), but it'll do for now. There isn't always a *right* way to clean up bad data.

Yes, you can use your models within the the code that changes your models.

Let's re-run the migration.

```
$ rake db:migrate
== 20160206002740 ChangeEmailNotNullInUser: migrating ========================
-- change_column(:users, :email, :string, {:null=>false})
   -> 0.0137s
== 20160206002740 ChangeEmailNotNullInUser: migrated (0.1069s) ===============
```

After my edits, the migration works. But there's one last thing to fix.

Whether or not you made a mistake, try to "rollback" this migration.

```
$ rake db:rollback
== 20160206002740 ChangeEmailNotNullInUser: reverting ========================
rake aborted!
StandardError: An error has occurred, this and all later migrations canceled:

ActiveRecord::IrreversibleMigration
```

If you had made a mistake, it's time to panic. It seems like you can't fix it.

When I use the `change` method, Rails can usually guess what should happen to undo my change.

- If I use `add_column`, Rails knows how to undo that change with `remove_column` (and vice versa)
- If I use `add_index`, Rails knows how to undo that change with `remove_index` (and vice versa).

When I make an edit like this to the `change` method, Rails can't figure out how to undo this change anymore. What's the opposite of setting some email values? Setting them back to `nil`? How would Rails know which ones to reset?

It doesn't.

Instead, it throws an error when I attempt to rollback this migration. I need to tell Rails what to do.

db/migrate/20140609051816_add_email_not_null_to_user.rb

```
class AddEmailNotNullToUser < ActiveRecord::Migration
  def up
    User.where(email: nil).each { |user|
      user.email = "#{user.name}@wyncode.co"
      user.save
    }
    change_column :users, :email, :string, null: false
  end

  def down
    change_column :users, :email, :string, null: true
  end
end
```

Instead of using the `change` method, I can use a combination of an `up` and a `down`. The `up` method tells Rails what to do to migrate up one version. The `down` method does the opposite. This way Rails doesn't have to do any guessing.

Rails' generated migrations are usually good enough. But sometimes you need to get in there and write some code to handle more complex cases. Rails can't do everything for you (but it tries).

And when you get in there and customize your migrations, it's good practice to make sure those migrations are reversible (just in case you mess it up).

We didn't actually check if this *not-null* database validation works.

Quiz: How would we do that?

Validations are Defensive Programming

Just like you need to protect your methods from bad arguments that can cause them to break, you need to protect your database from bad data that can cause your app to break.

Bad data is particularly harmful because it doesn't break right away. You store a bad email once, but you don't know it's bad until you try to use it. When you try to use it, it breaks. So you have to let the user know his email is bad. So you… send him an email?

Bad data is so painful to resolve that apps typically try to block it in three different places:

1. In the database (SQL), using these database column restrictions.
2. In ActiveRecord (Ruby), using validators.

3. In the browser (JavaScript), by making bad data in `<input>` fields glow bright colors when they're wrong.

It's wet, but bad data is so bad that sometimes this much defensiveness is necessary.

Review

- ActiveRecord models read their attributes from the database automagically - no need for `attr_accessor`
- Migrations can add, remove, and change columns and indices.
- When you name your migrations appropriately, Rails generators will write the code for you.
- When you make a mistake, rollback the migration first, then fix it, then migrate again.
- If you make a big mistake, drop and recreate the database from scratch (or get some help if you don't want to lose your data).
- Indices make your database faster and optionally enforce uniqueness.
- Uniqueness and presence can (and probably should) be enforced in the database, in Rails, and in JavaScript.

Associations

Skip this section. It was covered already in the new models 1 lecture. I'm keeping it here for my future reference.

So far we've only been working with a single model. Most apps compose more than one model. Let's create another model. Let's create a CodeSchool.

```
$ rails g model CodeSchool name
    invoke  active_record
    create    db/migrate/20140606205058_create_code_schools.rb
    create    app/models/code_school.rb
    invoke    test_unit
    create      test/models/code_school_test.rb
    create      test/fixtures/code_schools.yml
```

Each User should be associated with a CodeSchool. Let's ask Rails to create a migration to add a reference from User to CodeSchool.

```
$ rails g migration AddCodeSchoolToUser code_school:references
    invoke  active_record
    create    db/migrate/20140606205458_add_code_school_to_user.rb
```

db/migrate/20140606205458_add_code_school_to_user.rb

```
class AddCodeSchoolToUser < ActiveRecord::Migration
  def change
    add_reference :users, :code_school, index: true, foreign_key: true
  end
end
```

That's a little vague. What's a reference? Is this adding a column to the users table? If so, what kind of column? Let's run it and find out.

```
$ rake db:migrate
== 20140606205058 CreateCodeSchools: migrating ================================
-- create_table(:code_schools)
  -> 0.0117s
== 20140606205058 CreateCodeSchools: migrated (0.0119s) =======================

== 20140606205458 AddCodeSchoolToUser: migrating ==============================
-- add_reference(:users, :code_school, {:index=>true})
```

```
   -> 0.0091s
== 20140606205458 AddCodeSchoolToUser: migrated (0.0092s) ======================
```

We'll examine the User object in the Rails Console.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> User.columns.each { |c| puts c.type, c.name, "" }.length
integer
id

string
name

datetime
created_at

datetime
updated_at

string
email

integer
code_school_id

=> 6
```

I used the `columns` method to get a list of the column objects. Then I used `each` to `puts` some information about each of the columns.

You can get a similar result by just typing `User`. Usually you'll just check schema.rb, which will be updated for you with a Ruby representations of your latest database *schema* - meaning the overall structure of your tables, columns, and indices (DDL).

It turns out `add_reference` adds an integer column whose name is the same as the referenced class (with underscores) followed by "_id". How do we use it?

```
$ rails c
Loading development environment (Rails 4.1.1)
```

```
irb(main):001:0> c = CodeSchool.create(name: 'Wyncode')
…
u = User.take
irb(main):002:0> u.code_school_id = c.id
=> 1
irb(main):003:0> u.save
…
irb(main):004:0> u = User.find u.id
…
irb(main):005:0> u.code_school_id
=> 1
irb(main):006:0> u.id
=> 1
```

So I can set the code_school_id column, save it, and retrieve it. So that works. But it's kinda.. awkward. I keep having to refer to the "id" column everywhere. There must be a better way.

And there is.

app/models/user.rb

```
class User < ActiveRecord::Base
  belongs_to :code_school
end
```

The `belongs_to` method defines an association between the User object the the CodeSchool object. It does exactly what it says on the box. It defines an association where a User "belongs to" a CodeSchool. This line of code unlocks some new features.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> u = User.find 1
… stuff you've seen before and that's boring now ...
irb(main):002:0> c = u.code_school
  CodeSchool Load (0.4ms)  SELECT  "code_schools".* FROM "code_schools"  WHERE
"code_schools"."id" = $1 LIMIT 1  [["id", 1]]
=> #<CodeSchool id: 1, name: "Wyncode", created_at: "2014-06-06 21:10:04", updated_at:
"2014-06-06 21:10:04">
irb(main):003:0> c.name
=> "Wyncode"
irb(main):004:0> u.code_school_id
=> 1
```

With the `belongs_to` in place, instead of working with `_id` fields, I can reference objects. `User` still has a `code_school_id` field. That doesn't go away. But `belongs_to` adds a plain `code_school` field. And when I access it, I get a fully populated `CodeSchool` object.

These kinds of associations help me forget that there's a database behind the scenes. Instead, I can focus on the objects themselves.

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> u = User.find 1
…
```

I'm going to set the `code_school` object to `nil`, not the `code_school_id`.

```
irb(main):002:0> u.code_school = nil
=> nil
irb(main):003:0> u.save
…
```

I can confirm that the `code_school` and the `code_school_id` both updated accordingly.

```
irb(main):004:0> u = User.find 1
...
irb(main):005:0> u.code_school
=> nil
irb(main):006:0> u.code_school_id
=> nil
```

Now I can set the `code_school` field to a `CodeSchool` object and save it.

```
irb(main):007:0> u.code_school = CodeSchool.first
  CodeSchool Load (1.1ms)  SELECT  "code_schools".* FROM "code_schools"   ORDER
BY "code_schools"."id" ASC LIMIT 1
=> #<CodeSchool id: 1, name: "Wyncode", created_at: "2014-06-06 21:10:04", updated_at:
"2014-06-06 21:10:04">
irb(main):009:0> u.save
   (0.3ms)  BEGIN
  SQL (0.7ms)  UPDATE "users" SET "code_school_id" = $1, "updated_at" = $2 WHERE
"users"."id" = 1  [["code_school_id", 1], ["updated_at", "2014-06-06 21:40:25.873661"]]
   (0.8ms)  COMMIT
=> true
```

Note that the `CodeSchool` object is retrieved from the database with a SELECT, and it's `id` is set into the `User` record with an UPDATE.

I can confirm that both the `code_school` and the `code_school_id` fields updated accordingly.

```
irb(main):009:0> u = User.find 1
...
irb(main):010:0> u.code_school_id
=> 1
irb(main):011:0> u.code_school
...
=> #<CodeSchool id: 1, name: "Wyncode", created_at: "2014-06-06 21:10:04", updated_at:
"2014-06-06 21:10:04">
```

What if I have a `CodeSchool` object and I want to get its students. I told Rails that a `User` "belongs to" a `CodeSchool`. How do I tell Rails about the relationship in the other direction?

By telling Rails that a `CodeSchool` "has many" `Users`.

app/models/code_school.rb

```
class CodeSchool < ActiveRecord::Base
  has_many :users
end
```

```
$ rails c
Loading development environment (Rails 4.1.1)
irb(main):001:0> c = CodeSchool.first
  CodeSchool Load (1.0ms)  SELECT  "code_schools".* FROM "code_schools"   ORDER BY
"code_schools"."id" ASC LIMIT 1
=> #<CodeSchool id: 1, name: "Wyncode", created_at: "2014-06-06 21:10:04", updated_at:
"2014-06-06 21:10:04">
irb(main):002:0> c.users.count
   (0.5ms)  SELECT COUNT(*) FROM "users"  WHERE "users"."code_school_id" = $1
[["code_school_id", 1]]
=> 1
irb(main):003:0> u = c.users.first
=> #<User id: 1, name: "Ed", created_at: "2014-06-06 19:22:47", updated_at: "2014-06-06
21:40:25", email: "ed@wyncode.co", code_school_id: 1>
irb(main):004:0> u.name
=> "Ed"
```

With the `has_many` association defined, I can now get the `User` objects in a `CodeSchool`.

Note that I used the `count` method instead of the `length` method to get the total number of users.

```
$ rails c
CodeLoading development environment (Rails 4.1.1)
irb(main):001:0> cs = CodeSchool.take
  CodeSchool Load (1.3ms)  SELECT  "code_schools".* FROM "code_schools"  LIMIT 1
=> #<CodeSchool id: 1, name: "Wyncode", created_at: "2014-06-06 21:10:04", updated_at:
"2014-06-06 21:10:04">
irb(main):002:0> cs.users.length
  User Load (1.2ms)  SELECT "users".* FROM "users"  WHERE "users"."code_school_id"
= $1  [["code_school_id", 1]]
=> 1
irb(main):003:0> cs.users.count
   (0.5ms)  SELECT COUNT(*) FROM "users"  WHERE "users"."code_school_id" = $1
[["code_school_id", 1]]
=> 1
```

Note the difference in the generated SQL (in bold) for `length` and `count`. `length` pulls all the users from the database and counts them in Ruby. `count` tells MySQL to count the rows and just return the number. `count` is faster than `length`. Less data needs to be generated and sent over the wire.

Using associations, I can use models in a more natural, human-friendly way. I don't have to worry about the database. The `CodeSchool` associated with a `User` is located in the `User`'s `code_school` field. The `User`s associated with a `CodeSchool` are in the `CodeSchool`'s `users` field.

It's still kinda awkward to refer to the "users" in a school. They're not "users". They're "students". Should I rename my User class (and my users table) to Student instead? No need. I can just refer to the users by a different name.

app/models/code_school.rb

```
class CodeSchool < ActiveRecord::Base
  has_many :students, class_name: 'User'
end
```

When I told Rails that my `CodeSchool` `has_many` "users", it was able to figure out that I meant it has many `User` objects (via the naming convention). But if I'd like my `CodeSchool` to `has_many` "students", then Rails can't guess what I mean anymore. So I can tell Rails which class I'm referring to with the `class_name` option (once again, in an options Hash).

rails console

```
irb(main):001:0> CodeSchool.take.students
  CodeSchool Load (0.9ms)  SELECT  "code_schools".* FROM "code_schools"  LIMIT 1
  User Load (0.8ms)  SELECT "users".* FROM "users"  WHERE "users"."code_school_id" = $1
[["code_school_id", 1]]
=> #<ActiveRecord::Associations::CollectionProxy [#<User id: 1, name: "Ed", created_at:
"2014-06-06 19:22:47", updated_at: "2014-06-06 21:40:25", email: "ed@wyncode.co",
code_school_id: 1>]>
```

So `has_many` and `belongs_to` are on either end of the same association. A CodeSchool has_many Users and a User belongs_to a CodeSchool.

There are other types of associations as well. We'll talk about those later.