Time	3 hours	
Learning Goals	 Understand Object-Relational Mapping (ORM). Understand Active Record. Understand the relationship between SQL and Active Record. Understand why we use Active Record to implement an ORM instead of writing 	

Note: this is a loooong challenge with lots of reading to be done. Please don't rush through the challenge. Read it up carefully, explore the files that you will be cloning, and understand what's going on.

Object-Relational Modeling

In previous challenges we've drawn a map between Ruby World and SQL World that looked something like this:

SQL World	Ruby World
A table named `politicians`	A class named `Politician`
A row from `politicians`	An instance of `Politician`
SELECT * FROM politicians WHERE party = 'D'	Politician.where('party = ?', 'D')
SELECT * FROM politicians WHERE id = 10	Politician.where('id = ?', 10)

In SQL World we talk in terms of tables, rows, and relations. In Ruby World we talk in terms of classes, objects, and associations. The table above elucidates a kind of mapping that has a name: Object-Relational Mapping or ORM.

There are many libraries in Ruby which implement an ORM, including

- Active Record, which is what Rails uses by default
- Sequel
- Data Mapper

The idea behind most Ruby ORMs is that your model classes inherit from a base class that implements the general functionality to interact with the database and any model-specific code lives in the model (child) class. This can be tricky because the parent class

doesn't know beforehand anything about the structure of your database. How is it supposed to know that the class Student corresponds to the table students?

We're going to implement ActiveRecord, Jr. to get an idea of how one might build an ORM. This will make it much easier to reason about how the real ActiveRecord works once we get there, and will answer many of your questions about how to organize database calls in your code. Let's get started!

Clone the repository

First, fork the <u>challenge repository</u> and clone it. This code assumes there are two tables: students and cohorts. A student belongs to a single cohort and a cohort has many students.

This might be the most advanced app you've seen, but don't freak out! Here's what each file means:

Filename	Description
`app.rb`	Loads all of ActiveRecord, Jr. and our application-specific code
`Gemfile`	A [bundler](http://gembundler.com/) Gemfile, used to track the gems the app ne
`Gemfile.lock`	A misc. file related to the Gemfile
`models`	A directory containing all your model classes
`models/cohort.rb`	Contains the `Cohort` class
`models/database_model.rb`	Contains the `Database::Model` class, our "ActiveRecord, Jr."
`models/student.rb`	Contains the `Student` class
`Rakefile`	A list of [rake](http://en.wikipedia.org/wiki/Rake_%28software%29) tasks, used tasks like creating a database and seeding the database with dummy data

Filename	Description
`README.md`	The README! Sadly, empty right now. Use this challenge as the README and utterly confusing.

Besides the above, pay attention to the first line of each file. Pay extra attention if you see things like require_relative 'config/application' or something similiar. Where do these files link to and why is the require command needed?

Set up the Skeleton App

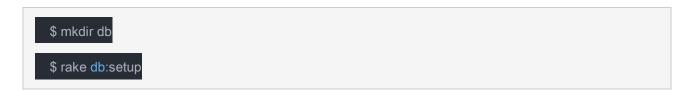
Run the following commands inside the activerecord_jr directory to start interacting with the code:

1. Run bundle

\$ bundle

This will read the Gemfile and install all the necessary gems for your app to run.

2. Set up the database



mkdir db will create an empty folder db so that when rake db:setup is called, a database is created in db/students.db and both students and cohorts tables are created in students.db database. Read the Rakefile if you're curious how this works.

3. Populate the database

\$ rake db:seed

This will populate your database with dummy cohort and student data. It will fail if you don't run rake db:setup first.

4. Playtime!

\$ rake console

This will drop you into an <u>IRB shell</u> with all your application code loaded and working. Try it out by running the below. (BTW, "Ramon" may not be a student in your database because it is dynamically generated by a rake task. You may need to try another common first name or peek in your .db file in sqlite3 to see who's there.)

```
ramons = Student.where('first_name = ?', 'Ramon')

ramons.first[:first_name]

cohort = Cohort.where('name = ?', 'Alpha').first

cohort.students.count

cohort.students.first[:email]
```

Write Simple Tests

Before we refactor, we should write some simple tests. The tests don't need to be super thorough, but it should verify the core functionality: reading/writing from/to the database, updating attributes, etc.

Once you have a handful of tests that pass for the skeleton code, you'll be more certain that your refactoring isn't inadvertently changing anything. If you refactor and the tests now fail, you'll know something has changed.

rake db:seed should continue to work the same before and after your refactoring, for example.

Write a script that you can run to sanity check your refactoring. Make sure, for example, that something like this continues to work:

```
cohort = Cohort.find(1)

cohort[:name] = 'Best Cohort Ever'

cohort.save
```

t This re-queries the database, so we're checking that we actually saved the data as intended

Cohort.find(1)[:name] == 'Best Cohort Ever'

You can create a simple test file called, e.g., test.rb, in the root directory of Active Record, Jr. It should include the app.rb on the first line, which automatically includes all the necessary code. Something like this:

require_relative 'app'

Your totally awesome test code goes her

When you run test.rb it should print out useful information so that it's easy as pie to tell when you've (accidentally) broken something.

Refactor Into the Base Class

There are three core files to this application:

- 1. models/database model.rb
- 2. models/student.rb
- 3. models/cohort.rb

If you look at student.rb and cohort.rb you'll see they have tons and tons of code in common. We're going to start by abstracting out the most simple of the common code into the base Database::Model class.

Be careful because the base class only knows what it's told. Database::Model doesn't know that Student maps to the students table, for example. Refactor out the following shared methods from Student and Cohort without doing anything to the Student and Cohort classes beyond removing the methods. The code in the methods might need to change, though.

1. Move Student#initialize and Cohort#initialize to Database::Model#initialize

- 2. Move Student#save and Cohort#save to Database::Model#save
- 3. Move Student#[] and Cohort#[] to Database::Model#[]
- 4. Move Student#[]= and Cohort#[]= to Database::Model#[]

Feel free to play around in the ActiveRecord, Jr. console to get a feel for how it works. And remember, once a method is moved to Database::Model you might need to change hard-coded references to the classes the method came from.

Submit Your Refactored Code

Add and commit the changes made to the repo. Then, git push it when you're done.

Questions for Reflection

- What did these refactorings accomplish?
- If we decided to add or remove a field in the database, how many changes on the code would you have had to make before? What about after? What about other changes to the databases?
- Using your new code, can you write a script to populate the database with dummy data?