RRC - Ecommerce - A2 Active Record

Assignment Two - Active Record

The purpose of this assignment is to exercise your Ruby and ActiveRecord skills. Active Record is the Ruby database library used by the Rails framework, but for this assignment you will be using it with pure Ruby.

In this assignment you will be working with a pre-prepared database for an ecommerce store by way of ActiveRecord. You will be using ActiveRecord to:

* Fix up some errors in the customer data.
* Add orders for specific customers.
* Generate invoices for customers with orders.

**Resources**

You may find the following resources helpful while coding this assignment:

* [The Ruby API](http://www.google.com/url?q=http%3A%2F%2Fwww.ruby-doc.org&sa=D&sntz=1&usg=AFQjCNGbsQLoGQBI8SKiAc-ZFuU_pk9Fqw)
* [Active Record Query Interface - Rails Guides](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Factive_record_querying.html&sa=D&sntz=1&usg=AFQjCNEPG8nPN8dQEWch5y1tWP24fFaF-Q)
* [Active Record Validations - Rails Guides](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Factive_record_validations.html&sa=D&sntz=1&usg=AFQjCNGxfO6sClNALuZoSM5H6U2R6yK4rA)
* [Active Record Associations - Rails Guides](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Fassociation_basics.html&sa=D&sntz=1&usg=AFQjCNFE6H8zPkAGv4cIzRZC5XcSsDmw2Q)

Also available:

* [Our In-Class Active Record Sample Code](https://www.google.com/url?q=https%3A%2F%2Fgithub.com%2Fstungeye%2FActiveRecord-without-Rails&sa=D&sntz=1&usg=AFQjCNHiloWlTXOpZRvtwIi34o-YvO_yjg)

Part Zero - Exploring the Provided Code

The code for this assignment will closely follow the structure we used for the Active Record challenge. The required starter code is provided as a zip file attached to the the Learn dropbox. Take a moment to read through the provided ar.rb file, the provided AR classes, and the demo code in customer\_list.rb.

Provided Folders and Files:

<root>

  |- ar.rb (Loads ActiveRecord plus all the provided AR classes.)

  |- customer\_list.rb (Demo that list details about 10 customers.)

  +- <app>

  |    +- <models> (The provided AR classes.)

  |         |- customer.rb

  |         |- line\_item.rb

  |         |- order.rb

  |         |- product.rb

  |         |- province.rb

  +- <db>

       |- development.sqlite3 (The sqlite3 database file.)

**Part One - Fixing Up Our Customers**

Let’s begin by inspecting a Customer object:

load 'ar.rb'

c = Customer.first

puts c.inspect

#<Customer id: 498, first\_name: "Christopher Hartman", last\_name: nil, address: "Ap #662-4186 Morbi Ave", city: "Lac La Biche County", country: "Canada", postal\_code: "J5V 8K0", email: "amet@nisiCum.co.uk", province\_id: 1, created\_at: "2011-11-16 05:31:31", updated\_at: "2011-11-16 05:31:31">

Notice how the entire name is stored in the first\_name property, while the last\_name property is nil?

In a file called customer\_name\_fix.rb write a short Ruby script that finds and fixes all customers such that the full name found in the first\_name property is split between the first\_name and last\_name properties.

Your script will need to:

* Find all customers in the customers table by way of the Customer class.
* Loop through each customer locating their full name in their first\_name property.
* Split the name on the first space you find. (Let’s assume that all customers have a single first name.)
* Modify each customer’s first\_name and last\_name property with the correct data.
* Save each customer back to the database.

Once you have fixed all your customers, modify the provided customer\_list.rb script, such that it will display the full customers name not only their first names.

Before you begin you may wish to make a backup of the development.sqlite3 database file.

**Part Two - Adding Orders and Line Items**

Take a moment to study the ERD diagram for our database in appendix B. Make note of the various foreign keys and the relationships between the tables. They can be defined as follows:

* Customers belong to a Province.
* A Province has many Customers.
* Orders belong to a Customer.
* A Customer has many Orders.
* An Order has many Products through LineItems.
* A Product has many Orders through LineItems.
* LineItems belong to an Order.
* An Order has many LineItems.
* LineItems belong to a Product.
* An Order has many Products.

BOOYA!

Check out how these relationships are defined in the AR classes in the app/models folder. You should then take a moment to read about [AR belongs to associations](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Fassociation_basics.html%23the-belongs_to-association&sa=D&sntz=1&usg=AFQjCNEhybuGpNPZIva8VV0krewvdPwO-Q), [AR has many associations](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Fassociation_basics.html%23the-has_many-association&sa=D&sntz=1&usg=AFQjCNERMbalDiAYPWiRmvcgYJmDzcPl7w), and [AR has many through associations](http://www.google.com/url?q=http%3A%2F%2Fguides.rubyonrails.org%2Fassociation_basics.html%23the-has_many-through-association&sa=D&sntz=1&usg=AFQjCNErkGexb4ZNqPIVoclqYJWrcFm5Cw).

Let’s create a new order for the first customer in our database:

customer = Customer.first

order = customer.orders.build

Inspect the order and you’ll see that the customer\_id foreign key is correctly set to match the id of the customer. Do it. Really. Type those lines into a file, add some puts statements to inspect the customer and the order. I’ll wait.

Now set one of the order properties and save the order:

order.status = 'new'

# We should also save a backup of the pst, gst and hst rate

# from the customers province into the order. Add that to this code.

order.save

Now let’s add a product to this order by way of a line\_item.

product1 = Product.first

line\_item1 = order.line\_items.build # Notice here how the LineItem   
                                   # objects are called line\_items.

line\_item1.product = product1

line\_item1.quantity = 12

line\_item1.price = product1.price # Keep a backup of the price at   
                                 # the time of the order.

line\_item1.save

Are you still with me? Keep typing this out. Don’t be lazy and copypasta either. Programming is as much as physical act as it is a mental one.

We can retrieve an array of a customer’s orders like this:

customer\_orders = some\_customer.orders

Okay, enough with the tutorial, it’s your turn:

Create a file called seed\_customer\_orders.rb that does the following:

* Create and persist orders for three customers. (Each customer should be from a different province.)
* Each order should involve at least four line items. (Use different products for each customer.)
* Ensure that you set the status of all orders to ‘new’
* Ensure that you backup the pst, gst and hst rates from the customer’s province into the appropriate properties of the order.
* Ensure that each order actually belongs to a customer by way of its foreign key.
* Ensure that each line item has a set quantity.
* Ensure that each line item is associated with a specific product by way of its foreign key.
* Ensure that each line item is associated with an order by way of its foreign key.
* Ensure that a backup of the associated product price is saved in each line item.
* Once you’ve done the above, pick one of the customers who already has an order and make them a second order.

Being a clever coder, you will want to program this part of the assignment in a way that automates the process of adding the line items and orders. Some form of automation is required to obtain full marks for this part.

**Part Three - Customer Invoices**

If you want to find an array of all the customers with new orders:

customers\_with\_orders = Customer.includes(:orders)

                                .where(orders: { status: 'new' })

Create a new file called generate\_customer\_invoices.rb such that it:

* Finds all customers with new orders.
* Generates an invoice for each customer.
* Invoices should be generated based on the required information and layout from assignment one.

Please note that unlike assignment one, we are now dealing with three potential taxes. Only display the taxes that are actually relevant to a customer. (i.e. The taxes where the rate isn’t zero.)

**Submission and Self Evaluation**

Please ensure that your scripts include a comment block at the top of the file which states your name and a brief description of the script.

You will submit your Ruby scripts within a zip file to the appropriate Learn Dropbox.

NOTE: Along with your scripts you must also submit a self-evaluation based on the specifications in appendix A. The self-evaluation should be completed as follows:

* Grab a copy of the [A2 self-evaluation Word document](https://docs.google.com/document/d/1W61GZfqGTLXpYh4Qo6huCmM-jC74hAFt7HHTM2XhxmM/edit?usp=sharing).
* Complete the self-evaluation according its list of instructions while referencing the requirements from appendix A.
* 5% will be deducted from your assignment if you do not submit a self-eval.

**Appendix A - Assignment Requirements**

Part One - Customer Name Fix

* All customers full names have been properly split amongst their first\_name and last\_name properties.
* Neither the first name, nor the last name should contain a space.
* All modified customer objects have been persisted back to the database.
* The customer\_list.rb file was correctly modified to display full customers names now that the customers data has been fixed.

Part Two - Adding Customer Orders

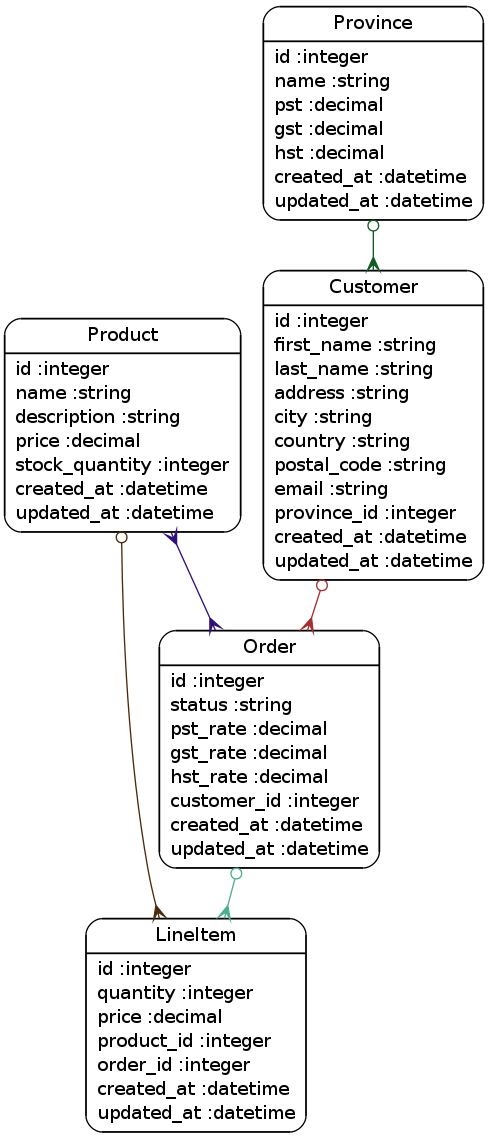
* Valid orders were created for at least three customers. (Validity as defined above.)
* At least one customers has two orders.
* Each order contains at least four valid line items. (Validity as defined above.)
* Product prices and customer tax rates are backed up in the appropriate locations.
* All associations between customers, orders, line items and products are defined.

Part Three - Customer Invoices

* An invoice is generated for each customer with an associated order.
* If a customer has multiple orders then multiple invoices are generated for that customer.
* Invoices follow the format of the invoices we generated in assignment one. The only difference is the now some customers will have HST instead of PST/GST.

General Requirements (Marks Deducted for Non Conformance)

* All code is nicely formatted and is easy to read.
* All nested code is indented using exactly 2 spaces.
* All variables and methods are named descriptively using snake case.
* All constants are named descriptively using screaming snake case.
* Complicated portions of your code are accompanied by comments that provide context and explanations.
* All files were named as specified.
* Your instructor reserves the right to deduct extra marks for any execution errors.

**Appendix B - The Database Schema ER**