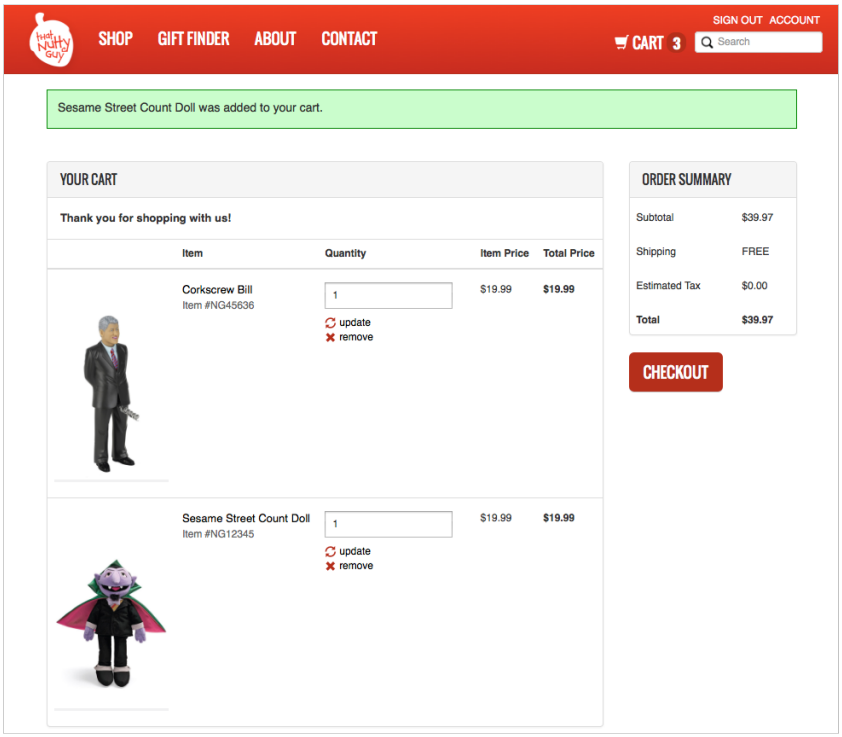
Exercise Preview



Exercise Overview

In this exercise, we'll get the cart working with the ability to add items to it. In Rails Level 1, we explored the most basic model relationships with **`has_many`** and **`belongs_to`.** Now we'll be looking at more complicated relationships between model objects.

1. If you completed the previous exercises, you can skip the following sidebar. We recommend you finish the previous exercises (2A–2D) before starting this one. If you haven't finished them, do the following sidebar.

If You Did Not Do the Previous Exercises (2A–2D)

1. Close any files you may have open.
2. On the **Desktop,** go to **Class Files > yourname-Rails Level 2 Class** and **delete** the **nutty** folder if it exists.
3. Select the **nutty-customizing active admin done** folder and hit **Cmd–D** to duplicate it.
4. **Rename** the folder to **nutty.**

3A Many-to-Many Relationships: Part One

Setup

1. On the Desktop, navigate to **Class Files > yourname-Rails Level 2 Class > That Nutty Guy HTML**
2. Open **cart.html** in a browser. This is what we'll be building now. Customers will be able to put products in the cart and then check out.
3. For this exercise, we'll continue working with the **nutty** folder located in **Desktop > Class Files > yourname-Rails Level 2 Class > nutty**

   If you haven't already done so, we suggest opening the **nutty** folder in your code editor if it allows you to (like Sublime Text does).
4. Launch Terminal.
5. In Terminal, type **cd** and a space, then do the following:
   - Drag the **nutty** folder from **Desktop > Class Files > yourname-Rails Level 2 Class** onto the Terminal window (so it will type out the path for you).
   - In Terminal, hit **Return** to change directory.
6. If you completed the previous exercises (2A–2D), skip this step and go on to the next one. If you started from a prepared folder, type the following into Terminal:

   ```
   bundle
   rake db:migrate
   rake db:seed
   ```
7. Run the Rails server in detach mode by typing the following:

   ```
   rails s -d
   ```
8. In a browser, go to: **localhost:3000** Feel free to check out the site. All the products should have their own images, and their specs should be nice, bulleted lists.

The Cart Controller

1. We usually like to start with a controller when implementing a new feature. In Terminal, type the following to create a cart controller:

   ```
   rails g controller cart
   ```
2. In your code editor, open **nutty > config > routes.rb**
3. A few lines above the **end** keyword, add the following code shown in bold:

   ```
   resources :products, only: [:index, :show]
   resources :cart, only: [:index, :create]

   root 'products#index'
   ```

4. Save the file, then close it.
5. In your code editor, open **nutty > app > controllers > cart_controller.rb.**
6. Add the following code shown in bold:

```
class CartController < ApplicationController
   def index
      @title = "Your Cart"
   end

   def create
   end
end
```

7. Save the file.

   The next step would typically be to create a view. It could take a lot of typing to set up all the design, so to make things easier for you, we've included a snippet with the code you need.

8. Open a Finder window and navigate to: **Desktop > Class Files > yourname-Rails Level 2 Class > snippets**
9. Click on the **index.html.erb** file and hit **Cmd–C** to copy it.
10. Still in the Finder, navigate to: **Desktop > Class Files > yourname-Rails Level 2 Class > nutty > app > views > cart**
11. Hit **Cmd–V** to paste the file into the **cart** folder.
12. In your code editor, open **nutty > app > views > cart > index.html.erb**

    Check out the code. Aren't you glad you didn't have to type all this?

13. In a browser, go to: **localhost:3000/cart** Looking good so far!
14. Click the **Cart** link at the top right. Notice that this took us to **cart.html.** We need to fix this.
15. In your code editor, open **nutty > app > views > layouts > application.html.erb**
16. Around line 43, find the link for **cart.html** and change it to **`/cart`** as shown below:

```
<a id="cart" href="/cart">
```

17. Save the file, then close it.
18. In a browser, go to **localhost:3000** (or reload it if you're already there).
19. Click the **Cart** link and now it should take you to **localhost:3000/cart**

    That's the bare minimum to get us up and running. The cart data we currently have is not being driven by Rails yet.

Making the Cart Visible Only to Registered Users

First, we know we're going to need a cart model. For simplicity's sake, let's say you have to be logged in to the site to add products to the cart.

1. Go to Terminal and create a cart model by typing:

   ```
   rails g model cart customer:references
   ```

   The reason we're adding **`customer:references`** is because we're going to require the cart to belong to a customer account. It's a way of making sure we only show the cart to the person it belongs to.

2. Apply it to the database by typing:

   ```
   rake db:migrate
   ```

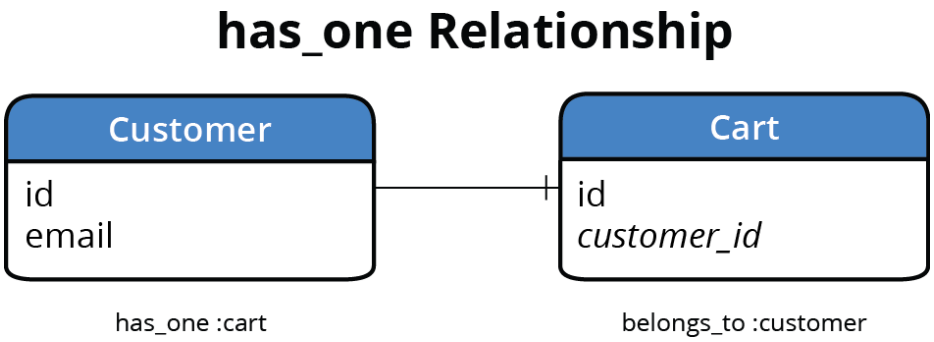
3. In your code editor, open **nutty > app > models > customer.rb**

4. Add the bold code to make sure the customer model knows about the cart:

   ```
      devise :database_authenticatable, :registerable,
             :recoverable, :rememberable, :trackable, :validatable

      has_one :cart
   end
   ```

   As shown in the diagram below, **`has_one`** is a way of saying that the Customer and Cart models are connected and the foreign key (`customer_id`) is in the Cart model. If you have a one-to-one relationship (as opposed to many-to-one), you would say `has_one` to refer from the model that doesn't have the foreign key (Customer) over to the one that does (Cart).



   NOTE: The diagram above is an entity-relationship model. The line connecting the two models shows the relationship between them. In this case, the `has_one` relationship is shown by using the following line —|—. In future diagrams in this workbook, you'll also see the following line —<— for a `has_many` relationship.

5. Let's next deal with loading the cart in the controller. Save the file, then close it.

6. In your code editor, open **nutty > app > controllers > cart_controller.rb**

7. We can only load the customer's cart if the customer is signed in. So let's check that before we go any further. We should also create a cart for the customer in case none exist. Add the bold code to specify what happens if a customer is signed in or not:

```
class CartController < ApplicationController
   def index
      @title = "Your Cart"
      if customer_signed_in?
         current_customer.create_cart if current_customer.cart.nil?
         @cart = current_customer.cart
      else
         redirect_to new_customer_session_path, alert: "Please sign in to
access your cart." and return
      end
   end
```

   The first part of the code checks if the customer is signed in. If they don't have a cart, one will be created for them. If the customer is not signed in, they will see an alert and be redirected to sign in. **`and return`** can be very important because otherwise, even though the customer was redirected, any code beneath it would continue executing.

8. Save the file.
9. In a browser, go to: **localhost:3000**
10. Click on the **Cart** link at the upper right.
11. In order to sign in, you'll need to create an account. Click the **Sign up** link.
12. Enter an email and password, then click **Sign Up.**
13. After signing in, go to the **Cart** page. You should have no problem viewing it now.

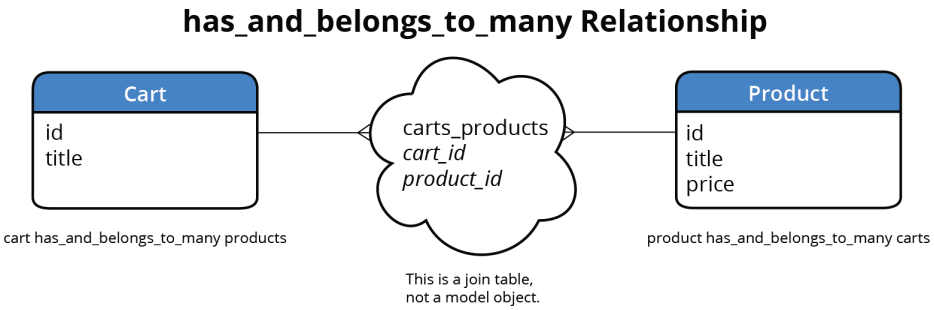
Many-to-Many Relationships

We don't yet have a way to associate products with the Cart model. `has_many` and `belongs_to` are not going to be enough for us here. For one thing, where would the foreign key go? If the cart id key went in the products table, then each product could only ever be put in one cart, which obviously won't work if we want to sell more than one! On the other hand, putting the foreign key, like a product id key, in the Cart model would limit us to one product per customer. What we need to implement here is a many-to-many relationship. Rails provides a number of tools for us to do just that.

NOTE: For a refresher of the `has_many` and `belongs_to` relationship with an example of a situation in which that model relationship would be suitable, read the sidebar at the end of the exercise.

3A Many-to-Many Relationships: Part One

The most basic kind of many-to-many relationship is **has_and_belongs_to_many.** This relationship allows each Product to have any number of carts, and each Cart to have any number of products in it. To implement this, we have to manually create a join table. A **join table** is a special database table, which keeps track of which products and which carts are associated.



1. Let's create a migration to do that. In Terminal, type:

   ```
   rails g migration create_carts_products cart:references product:references
   ```

   NOTE: The convention for join tables in Rails is that they're named alphabetically by model in the plural. So if you're relating cart and product, "c" comes before "p" and make them both plural.

2. Apply the migration by typing:

   ```
   rake db:migrate
   ```

3. In your code editor, open **nutty > app > models > cart.rb.**

4. Add the following bold code to let the cart model know about this relationship:

   ```
   class Cart < ActiveRecord::Base
      belongs_to :customer

      has_and_belongs_to_many :products
   end
   ```

5. Save the file, then close it.

6. In your code editor, open **nutty > app > models > product.rb.**

7. We also need to let the product model know about this relationship:

   ```
   validates :price, numericality: true

   has_and_belongs_to_many :carts

   has_attached_file :image, styles: { medium: "700x900>", thumb: "120x80>" },
   default_url: "/images/:style/missing.png"
   ```

   That's enough to create a many-to-many relationship between those two model objects.

8. Save the file and then close it.

Making the Add to Cart Button Functional

Let's turn the Add To Cart button into a proper form so we can put items in the cart.

1. In a browser, go to: **localhost:3000**
2. Click on **Corkscrew Bill.**
3. Click the **Add To Cart** button. Of course it doesn't work yet, but that's what we'll get working next.
4. In your code editor, open **nutty > app > views > products > show.html.erb**
5. Around line 25, add the following bold code:

```
<p class="star-rating-large five ir">Star Rating</p>
<%= form_tag '/cart', method: :post do %>

<% end %>

<input type="number" name="quantity" min="1" max="100" value="1">
```

NOTE: Posting to the root of a controller's route calls the `create` method in cart_controller.

6. Add the following bold code:

```
<%= form_tag '/cart', method: :post do %>
   <%= hidden_field_tag :product_id, @product.id %>
   <%= number_field_tag :quantity, 1, min: 1, max: 100 %>
   <%= button_tag id: 'add-cart', class: 'btn btn-red btn-md' do %>
      <span class="glyphicon glyphicon-shopping-cart"></span> Add to Cart
   <% end %>
<% end %>
```

NOTE: The **`hidden_field_tag`** lets the form know which products go in the cart. The **`number_field_tag`** creates a field for the quantity. The **`button_tag`** is our submit button.

7. Delete the old cart button code shown in bold below (around lines 32–37):

```
<input type="number" name="quantity" min="1" max="100" value="1">
<a href="cart.html">
   <button type="button" id="add-cart" class="btn btn-red btn-md">
      <span class="glyphicon glyphicon-shopping-cart"></span> Add to Cart
   </button>
</a>
```

8. Save the file.

Making Sure Customers Are Signed In

1. In your code editor, open **nutty > app > controllers > cart_controller.rb**

   Before a customer can add anything to their cart, they need to have a cart. Before they can have a cart, they need to be signed in. We could check if they are signed in by adding code to the `create` method… but that wouldn't be very DRY because we would be repeating code.

2. Instead, cut **(Cmd–X)** the following bold code:

```
class CartController < ApplicationController
   def index
      @title = "Your Cart"
      if customer_signed_in?
         current_customer.create_cart if current_customer.cart.nil?
         @cart = current_customer.cart
      else
         redirect_to new_customer_session_path, alert: "Please sign in to
access your cart." and return
      end
   end
```

3. Between the **`create`** method and the final **`end`** keyword, create a private (internal controller) method as shown in bold below:

```
   def create
   end

   private
      def load_cart_or_redirect_customer

      end
end
```

4. Paste **(Cmd–V)** the code you cut into the method as shown below:

```
private
   def load_cart_or_redirect_customer
      if customer_signed_in?
         current_customer.create_cart if current_customer.cart.nil?
         @cart = current_customer.cart
      else
         redirect_to new_customer_session_path, alert: "Please sign in to
access your cart." and return
      end
   end
```

   NOTE: Remember that private controller methods don't necessarily lead to a page—they are just called internally by the controller.

5. We want the private method to run before the `index` or `create` pages so the sign in code will run before customers view the cart or add anything to it. Add this code:

```
class CartController < ApplicationController
   before_action :load_cart_or_redirect_customer

   def index
```

Adding Products to the Cart

1. Now we can load up the cart with products. Around line 9, type:

```
def create
   product = Product.find(params[:product_id])
   @cart.products << product
   @cart.save
   redirect_to '/cart', notice: "#{product.title} was added to your cart." and
return
end
```

   NOTE: Remember, from show.html.erb, we're going to pass ourselves the product's id as a hidden field so we will get it back in the `params` hash. The product is then added to the cart. Then the cart is saved. Finally the customer is redirected to the cart page with a notice that the product was added successfully.

2. Save the file.

3. Let's try it out and see if it works. In the browser, go back to the **Corkscrew Bill** page and reload it.

4. Click **Add To Cart.**

   It's working, but our cart is still showing the sample data. We need to implement the actual items in the cart.

5. In your code editor, open **nutty > app > views > cart > index.html.erb**

6. Around lines 20 and 36, wrap the existing code in the following bold tags:

```
<tbody>
   <% @cart.products.each do |product| %>
      <tr>
         <td id="thumbnail-div" class="hidden-xs">

         CODE OMITTED TO SAVE SPACE

         <td class="hidden-xs">$19.99</td>
         <td class="total-price">$19.99</td>
      </tr>
   <% end %>
      <td id="thumbnail-div" class="hidden-xs">
```

7. We only need one row to make the cart show the correct item(s). Before we make that row dynamic, first **delete** the extra row (around lines 37–49):

```
<td id="thumbnail-div" class="hidden-xs">
   <a href="#"><img src="img/product_images/toothbrush.jpg" class="cart-
thumbnail" alt="Pantone Toothbrush"></a>
</td>
<td>
   <a href="#"><p>Pantone Toothbrush Set</p></a>
   <p class="gray-text">Item #NG00921</p>
</td>
<td><input type="number" name="quantity" min="1" max="100" value="2">
   <a href="#"><span class="glyphicon glyphicon-refresh"></span>update</a>
   <a href="#"><span class="glyphicon glyphicon-remove"></span>remove</a>
</td>
<td class="hidden-xs">$9.99</td>
<td class="total-price">$19.98</td>
```

8. Around line 23, highlight the link and image:

```
<a href="product.html"><img src="img/product_images/corkscrew-bill.jpg"
class="cart-thumbnail" alt="Corkscrew Bill"></a>
```

9. Delete it and replace it with the bold code shown below:

```
<td id="thumbnail-div" class="hidden-xs">
   <%= link_to product do %>
      <%= image_tag product.image.url(:medium), alt: product.title, class:
'cart-thumbnail' %>
   <% end %>
</td>
```

10. Around line 28, find the following code and highlight it:

```
<a href="product.html"><p>Corkscrew Bill</p></a>
```

11. Replace it with the bold code as shown:

```
<td>
   <%= link_to product do %>
      <p><%= product.title %></p>
   <% end %>
   <p class="gray-text">Item #NG45636</p>
```

12. Replace the item number as shown (around line 31):

```
   <p class="gray-text">Item #<%= product.sku %></p>
```

We'll implement the quantity, update, and remove actions later.

13. Replace the product prices ($19.99) as shown (around lines 37–38):

```
<td class="hidden-xs"><%= number_to_currency product.price %></td>
<td class="total-price"><%= number_to_currency product.price %></td>
```

14. Save the file.

15. Go back to the cart in the browser and reload it: **localhost:3000/cart**

    Now it should only have Corkscrew Bill in it.

16. Try adding another product. Click on the logo at the top left to go home.

17. Click any product to go to its page. Then click the **Add To Cart** button.

    Voilà! Now we should have the two items we added to our cart!

How Do We Implement the Quantity Field?

1. While we're still looking at the cart in the browser, take a look at the **Quantity** field. We haven't done anything to make it functional yet.

   How do we implement this Quantity field? Does it become a property of the product? That wouldn't make sense. Does it become a property of the cart?

   As we strategize what to do about the Quantity field, we begin to see the problem with the `has_and_belongs_to_many` relationship. It's sufficient for the most basic many-to-many relationships. However, at some point in nearly any implementation, you'll find that you need to tack on some additional metadata to the relationship between two models.

   The Quantity field refers to the relationship of the product and the cart. So we somehow need to get the quantity into the join table we created in this exercise. With a simple join table that isn't a model object of its own, there's no room there to do it.

   In the next exercise, we'll look at a better way to handle these kinds of relationships.

2. Leave your code editor and browser open, as well as the server running in Terminal so we can continue with them in the following exercises.

The has_many & belongs_to Relationship

One of the most basic model relationships in Ruby on Rails is one in which one model **has_many** instances of objects from another model (which **belongs_to** the model that `has_many`). It's just like the **has_one** relationship introduced earlier in this exercise, except that objects from one model will need to be associated with **more than one** object (even a lot of them!) from the connected model.

The diagram below references a movie database app that has two connected models. Each **movie** in the app has a unique numeric **id** field, which indicates that each movie only appears once in the database. That's your indicator that the Movie model is the one that `has_many`.

In this app, the goal is to give the site's admins the ability to assign as many different actors **(cast members)** to each movie as the cast roster calls for. To create that functionality, we would make sure the Cast Member model **belongs_to** the Movie model. The Cast Member model gets a **foreign key** which says that it belongs to the Movie model, the one that `has_many`.

The foreign key **movie_id** refers to an integer field that stores the unique number associated with each item in the Movie model. This field stores the id of the movie each cast member is associated with. It is called "foreign" because it refers to another table, and called a "key" because it refers to that table's primary key (id).