28. Creating Your Own Shopping Cart

In this chapter you get to look at another practical implementation example of a shopping cart. The reason that a shopping cart example was chosen is because it is very common and familiar so the concepts will be easy to pick up on throughout the chapter. Also the shopping cart example will allow us to look at one way to utilize the ability to switch between views in AngularJS as you navigate through the checkout process.

The shopping cart you will create provides most of the necessary functionality, however, it is not intended for production. There is missing validation and error handling in areas to make the project reasonable to fit inside a book example. However, the project will take you through the basic steps and provide you with fundamental understanding of how the Node.JS, MongoDB and AngularJS operates.

Project Description

The implementation in this chapter is a fairly basic shopping cart that allows you to add items, remove items and then go through the checkout process and view your orders. The example provides most of the basic functionality that is required for a shopping cart, however, it is not intended for production use but rather as a teaching aid.

Keep in mind that the cart does not actually link to a credit card verification service to process payments. That goes beyond the scope of the project. There is still a fair amount of work to take this project and make it into something that could be used in production.

Also login, authentication and session management are also omitted from the project since that was already covered in Chapter 26 and would add additional code to sift through to understand the shopping cart example. The example is hard coded to a user with a `userid` of `customerA`. You will see that places where the authenticated user would be used `customerA` is hard coded.

The general logic flow of the example is as follows:

1. The root page contains a list of prints that can be purchased.

2. The user clicks on a print to view details where they can add it to the cart. Multiple items can be added to the cart. The top link to the cart shows the number of items in the cart.

3. The user clicks on the cart and can see the items. They can change the quantity, remove items, continue shopping or checkout.

4. On checkout the customer is presented with shipping information page.

5. Then the customer is presented with billing information including the billing address and credit card data.

6. When the customer clicks to submit the billing information a review of the validated transaction is displayed and the user can select the final purchase.

7. After the final purchase is clicked the user is shown a list of their orders including the recently completed order.

Libraries Used

The project in this chapter uses the following add on Node.js NPM modules. You will need to install them into your project directory if you intend to follow along with the example code:

- **`express`:** Used as the main web server for the project.
- **`ejs`:** Used to render the HTML templates.
- **`mongodb`:** Used to access the MongoDB database.
- **`mongoose`:** Used to provide the structured data model.

This code in this chapter also requires the AngularJS library to be provided as well.

Project Directory Structure

The project is organized into the following directory structure to show you one method of organizing your code. You do not have to follow this directory structure, however, directory structure should be part of the overall design of your projects so that you can easily find the code that you are looking for.

- **`./`:** Project root that contains the base application files and supporting folders.
- **`./npm_modules`:** Created when the NPMs listed above are installed in the system.
- **`./controllers`:** Contains the Express route controllers that provide the interaction between routes and changes to the MongoDB database.
- **`./models`:** Contains the Mongoose model definitions for objects in the database.
- **`./static`:** Contains any static files that need to be sent such as CSS and AngularJS JavaScript and AngularJS partial templates.
- **`./views`:** Contains the HTML templates that will be rendered by EJS.
- **`../images`:** Contains any images for the project. This is at a peer level to the project so that the same images can be used by multiple projects.
- **`../lib`:** Contains the AngularJS libraries so that they can be loaded locally my multiple projects. In production you may want to use a CDN delivery address instead.

In addition to the directory structure the following code files are included. The following list is intended to give you an idea of the functionality of each file:

- **`./cart_init.js`:** Standalone initialization code for this example. It will create a single customer named customerA and add several products to use in the example.
- **`./cart_server.js`:** Main application file that loads the necessary libraries, creates a connection to MongoDB and starts the Express server.
- **`./cart_routes.js`:** Defines the routes for the Express server. Functionality that does not interact with the database is also handled directly in this file.
- **`./controllers/customers_controller.js`:** Defines the functionality for the routes that require interaction with the MongoDB database to get and update customer data such as cart, shipping and billing information.
- **`./controllers/orders_controller.js`:** Defines the functionality for the routes that require interaction with the MongoDB database to get the order objects.
- **`./controllers/products_controller.js`:** Defines the functionality for the routes that require interaction with the MongoDB database to get the one or all product objects.

• **`./models/cart_model.js`**: Defines the customer, product, order, and supporting models for the cart example.
• **`./views/shopping.html`:** Provides the main shopping page for the application that will allow the user to view products and place them into the cart.
• **`./static/billing.html`:** This is an AngularJS partial that implements the billing information view.
• **`./static/cart.html`:** This is an AngularJS partial that implements the shopping cart view.
• **`./static/orders.html`:** This is an AngularJS partial that implements a view with a list of orders.
• **`./static/products.html`:** This is an AngularJS partial that implements a list of products view.
• **`./static/product.html`:** This is an AngularJS partial that implements a view for a single product with an add to cart link.
• **`./static/review.html`:** This is an AngularJS partial that implements a view to allow the user to review the order before placing it.
• **`./static/shipping.html`:** This is an AngularJS partial that implements the shipping information view.
• **`./static/js/cart_app.js`**: Provides the AngularJS module and controller definitions to handle all of the shopping cart interaction between the AngularJS code and the webserver.
• **`./static/css/cart_styles.css`**: Provides the CSS styling for the AngularJS HTML pages.

Defining the Customer, Product and Orders Models

As always you should look at the object model needs as the first step in implementing your applications. In this case the object of the exercise is to provide a shopping cart. To do so we will need a customer model as a container for the shopping cart. Also we will need products to place into the shopping cart. To checkout we will need billing information and shipping information and then once the order is placed we will need a way to store the order.

Therefore we need a model for the customer, product, order billing and shipping data to support the shopping cart. The following sections discuss the design of each of the model schemas implemented in the project. The schemas are all contained in a file name `cart_model.js` that

implements all of the schemas. The first two lines and the final line of that file are as follows to load mongoose Schema:

```
01 var mongoose = require('mongoose'),
02     Schema = mongoose.Schema;
```

The rest of the lines will be given in each section below as they are described.

Defining the Address Schema

We begin by defining an address schema that will be generic so that it can be used in both the shipping information and as part of the billing information. The code in Listing 28.1 implements the `AddressSchema` containing the standard address information. Notice that the following code is used to disable the `_id` property since we will not need to look up addresses by id:

```
{ _id: false }
```

Listing 28.1. cart_model.js-AddressSchema Defining a Schema for Shipping and Billing Addresses

```
03 var AddressSchema = new Schema({
04   name: String,
05   address: String,
06   city: String,
07   state: String,
08   zip: String
09 }, { _id: false });
10 mongoose.model('Address', AddressSchema);
```

Defining the Billing Schema

With the `AddressSchema` defined we can now define the billing schema to keep track of credit card information as well as billing information. The code in Listing 28.2 implements the `BillingSchema` containing standard credit card data. Notice that the following implementation for `cardtype` will require that an entry of `Visa`, `MasterCard` or `Amex` is used, no other values are allowed:

cardtype: { type: String, enum: ['Visa', 'MasterCard', 'Amex'] },

Also notice that the `address` field is assigned the `AddressSchema` object type. Mongoose requires that nesting schemas in this way be included in an array. This is used in more than one place and you will see that through the example that the address will be access using `address[0]` to get the first item in the array.

Listing 28.2. cart_model.js-BillingSchema Defining a Schema for Billing Credit Card and Address

```
11 var BillingSchema = new Schema({
12   cardtype: { type: String, enum: ['Visa', 'MasterCard', 'Amex'] },
13   name: String,
14   number: String,
15   expiremonth: Number,
16   expireyear: Number,
17   address: [AddressSchema]
18 }, { _id: false });
19 mongoose.model('Billing', BillingSchema);
```

Defining the Product Schema

Next we define the schema to store product information the product model for this example is a print with a name, `imagefile`, `description`, `price` and `instock` counter. Listing 28.3 shows the full definition of the `ProductSchema`.

Listing 28.3. cart_model.js-ProductSchema Defining a Basic Product Schema for Prints

```
20 var ProductSchema = new Schema({
21   name: String,
22   imagefile: String,
23   description: String,
24   price: Number,
25   instock: Number
26 });
27 mongoose.model('Product', ProductSchema);
```

Defining the Quantity Schema

For orders and the shopping cart we can include the products in an array, however, we also need the ability to store a quantity. One method of doing this is to define a quantity schema with quantity and product fields. Listing 28.4 implements the `ProductQantitySchema` that does just that. Notice that since we are embedding the `ProductSchema` we include it as an array and through the example will access it using `product[0]`.

Listing 28.4. cart_model.js-QuantitySchema Defining a Basic Quantity Schema for Quantity of Products in Orders and the Cart

```
27 mongoose.model('Product', ProductSchema);
28 var ProductQuantitySchema = new Schema({
29   quantity: Number,
30   product: [ProductSchema]
31 }, { _id: false });
32 mongoose.model('ProductQuantity', ProductQuantitySchema);
```

Defining the Order Schema

Next is the order schema that keeps track of the items ordered, shipping information and billing information. Listing 28.5 implements the `OrderSchema` model that stores the necessary information about the order. Notice that a `Date` type is assigned to the `timestamp` field to keep track of when the order was placed. Also the `items` field is an array of `QuantityShema` sub documents.

Listing 28.5. cart_model.js-OrderSchema Defining the Order Schema to Store the Order Information

```
33 var OrderSchema = new Schema({
34   userid: String,
35   items: [ProductQuantitySchema],
36   shipping: [AddressSchema],
37   billing: [BillingSchema],
38   status: {type: String, default: "Pending"},
39   timestamp: { type: Date, default: Date.now }
40 });
41 mongoose.model('Order', OrderSchema);
```

Defining the Customer Schema

The final schema is the customer schema. Listing 28.6 implements the `CustomerSchema` that contains a unique `userid` field to identify the customer and associate orders with the customer. This field would normally map to an authenticated session `userid`.

Notice that the shipping, billing and cart are all nested schemas. That makes is simple to define the model. You will see that each of these will be accessed in the JavaScript using the `shipping[0]`,`billing[0]` and `cart[0]` array indexing. No cart object schema is necessary because it is inherently implemented by the array of `ProductQuantitySchema` subdocuments.

Listing 28.6. cart_model.js-CustomerSchema Defining the Customer Schema to Store the Shipping, Billing and Cart

```
42 var CustomerSchema = new Schema({
43   userid: { type: String, unique: true, required: true },
44   shipping: [AddressSchema],
45   billing: [BillingSchema],
46   cart: [ProductQuantitySchema]
47 });
48 mongoose.model('Customer', CustomerSchema);
```

Creating the Shopping Cart Server

With the model defined you can begin implementing the shopping cart server. The code in Listing 28.7implements the Express server for the shopping cart application. This code should be familiar to you. It includes the express and mongoose libraries and then connects to MongoDB via Mongoose.

Notice that there is a `require` statement for the model definition to build the `Schema` object within Mongoose. Also the `./cart_routes` files is included to initialize the routes for the server before listening on port 80.

Listing 28.7. cart_model.js Implementing the Shopping Cart Application Server Using Express and Connecting to MongoDB

```
01 var express = require('express');
02 var mongoose = require('mongoose');
03 var db = mongoose.connect('mongodb://localhost/cart');
```

```
04 require('./models/cart_model.js');
05 var app = express();
06 app.engine('.html', require('ejs').__express);
07 app.set('views', __dirname + '/views');
08 app.set('view engine', 'html');
09 app.use(express.cookieParser());
10 app.use(express.bodyParser());
11 require('./cart_routes')(app);
12 app.listen(80);
```

Implementing Routes to Support Product, Cart and Order Requests

As part of the Express server configuration the `./cart_routes.js` file shown in Listing 28.7 was loaded. The code in Listing 28.8 provides the routes necessary to get the customer, product and order objects. They also provide routes to add orders to the database and update the customer shipping, billing and cart information.

Lines 6-9 implement the static routes to support getting the AngularJS, CSS, JavaScript, images and AngularJS template partials used in this example. The images and AngularJS lib folders are located in a sibling directory to the project. The other static files are in the `./static` folder inside the project.

Notice that when the user access the root location of `/` for the site that the `shopping.html` template is rendered on line 11. The remaining routes on lines 13-19 all involve interaction with the MongoDB database and will be handled in the controller route handlers described in the next section.

Listing 28.8. cart_routes.js Implementing the Routes for Shopping Cart Web Requests from the Client

```
01 var express = require('express');
02 module.exports = function(app) {
03   var customers = require('./controllers/customers_controller');
04   var products = require('./controllers/products_controller');
05   var orders = require('./controllers/orders_controller');
06   app.use('/static', express.static( './static')).
07       use('/images', express.static( '../images')).
08       use('/lib', express.static( '../lib')
```

```
09   );
10   app.get('/', function(req, res){
11    res.render('shopping');
12   });
13   app.get('/products/get', products.getProducts);
14   app.get('/orders/get', orders.getOrders);
15   app.post('/orders/add', orders.addOrder);
16   app.get('/customers/get', customers.getCustomer);
17   app.post('/customers/update/shipping', customers.updateShipping);
18   app.post('/customers/update/billing', customers.updateBilling);
19   app.post('/customers/update/cart', customers.updateCart);
20 }
```

Implementing the Model Based Controller Routes

In addition to the standard route implementation you also need to implement route handling that interact with the database. These route handlers are broken out of the standard `cart_route.js` file and into their own files based on model to keep the code clean and ensure a good division of responsibilities.

The following sections cover the implementation of the model specific controllers for the `Customer`,`Order` and `Product` models.

Implementing the Product Model Controller

The code in Listing 28.9 implements the route handling code for the `Product` model. There are only two routes. The `getProduct()` route finds a single product based on the `productId` included in the query. The `getProducts()` route finds all products. If successful the product or all products are returned to the client as JSON strings. If the requests fail then a 404 error is returned.

Listing 28.9. products_controller.js Implementing the Get Product and Get Products Routes for the Express Server

```
01 var mongoose = require('mongoose'),
02     Product = mongoose.model('Product');
03 exports.getProduct = function(req, res) {
04   Product.findOne({ _id: req.query.productId })
```

```
05   .exec(function(err, product) {
06     if (!product){
07       res.json(404, {msg: 'Photo Not Found.'});
08     } else {
09       res.json(product);
10     }
11   });
12 };
13 exports.getProducts = function(req, res) {
14   Product.find()
15   .exec(function(err, products) {
16     if (!products){
17       res.json(404, {msg: 'Products Not Found.'});
18     } else {
19       res.json(products);
20     }
21   });
22 };
```

Implementing the Order Model Controller

The code in Listing 28.10 implements the route handling code for the `Order` model. There are three routes. The `getOrder()` route finds a single order based on the `orderId` included in the query. The`getOrders()` route finds all orders that belong to the current user. In the case of the example the`userid` of `customerA` is hard coded. If successful the order or all of this customer's orders are returned to the client as JSON strings. If the requests fail then a 404 error is returned.

The `addOrder()` route handler builds a new `Order` object by getting the `updatedShipping`,`updatedBilling` and `orderItems` parameters from the `POST` request. If the order saves successfully then the cart field in the customer object is reset to empty using the following code and a success is returned otherwise a 500 or 404 error is returned.

```
37       Customer.update({ userid: 'customerA' },
38           {$set:{cart:[]}})
```

Listing 28.10. orders_controller.js Implementing the Get Order(s) and Add Order Routes for the Express Server

```
01 var mongoose = require('mongoose'),
02     Customer = mongoose.model('Customer'),
03     Order = mongoose.model('Order'),
04     Address = mongoose.model('Address'),
05     Billing = mongoose.model('Billing');
06 exports.getOrder = function(req, res) {
07   Order.findOne({ _id: req.query.orderId })
08   .exec(function(err, order) {
09     if (!order){
10       res.json(404, {msg: 'Order Not Found.'});
11     } else {
12       res.json(order);
13     }
14   });
15 };
16 exports.getOrders = function(req, res) {
17   Order.find({userid: 'customerA'})
18   .exec(function(err, orders) {
19     if (!orders){
20       res.json(404, {msg: 'Orders Not Found.'});
21     } else {
22       res.json(orders);
23     }
24   });
25 };
26 exports.addOrder = function(req, res){
27   var orderShipping = new Address(req.body.updatedShipping);
28   var orderBilling = new Billing(req.body.updatedBilling);
29   var orderItems = req.body.orderItems;
30   var newOrder = new Order({userid: 'customerA',
31                     items: orderItems, shipping: orderShipping,
32                     billing: orderBilling});
33   newOrder.save(function(err, results){
34     if(err){
35       res.json(500, "Failed to save Order.");
36     } else {
```

```
37      Customer.update({ userid: 'customerA' },
38          {$set:{cart:[]}})
39      .exec(function(err, results){
40        if (err || results < 1){
41        res.json(404, {msg: 'Failed to update Cart.'});
42        } else {
43         res.json({msg: "Order Saved."});
44        }
45       });
46     }
47   });
48 );
```

Implementing the Customer Model Controller

The code in Listing 28.11 implements the route handling code for the `Customer` model. There are four routes.

The `getCustomer()` route finds a customer order based on the hard coded `customerA` value. If successful the customer object is returned to the client as JSON strings. If the requests fails then a 404 error is returned.

The `updateShipping()` route will create a new `Address` object from the `updatedShipping`parameter in the `POST` request and then use an `update()` method to update the customer object with the new shipping data. If successful a success message is returned, if it fails then a 404 error is returned.

The `updateBilling()` route will create a new Billing object from the `updatedBilling` parameter in the `POST` request and then use an update() method to update the customer object with the new billing data. If successful a success message is returned, if it fails then a 404 error is returned.

The `updateCart()` route will use the `update()` method to update the cart field of the customer with the `updatedCart` object sent in the `POST` request. If successful a success message is returned, if it fails then a 404 error is returned.

Listing 28.11. customers_controller.js Implementing the Customer Get and Update Routes for the Express Server

```
01 var mongoose = require('mongoose'),
02     Customer = mongoose.model('Customer'),
03     Address = mongoose.model('Address'),
04     Billing = mongoose.model('Billing');
05 exports.getCustomer = function(req, res) {
06   Customer.findOne({ userid: 'customerA' })
07   .exec(function(err, customer) {
08     if (!customer){
09       res.json(404, {msg: 'Customer Not Found.'});
10     } else {
11       res.json(customer);
12     }
13   });
14 };
15 exports.updateShipping = function(req, res){
16   var newShipping = new Address(req.body.updatedShipping);
17   Customer.update({ userid: 'customerA' },
18       {$set:{shipping:[newShipping.toObject()]}})
19   .exec(function(err, results){
20     if (err || results < 1){
21     res.json(404, {msg: 'Failed to update Shipping.'});
22     } else {
23      res.json({msg: "Customer Shipping Updated"});
24     }
25   });
26 };
27 exports.updateBilling = function(req, res){
28   // This is where you could verify the credit card information
29   // and halt the checkout if it is invalid.
30   var newBilling = new Billing(req.body.updatedBilling);
31   Customer.update({ userid: 'customerA' },
32       {$set:{billing:[newBilling.toObject()]}})
33   .exec(function(err, results){
34     if (err || results < 1){
35      res.json(404, {msg: 'Failed to update Billing.'});
36     } else {
```

```
37     res.json({msg: "Customer Billing Updated"});
38    }
39   });
40 };
41 exports.updateCart = function(req, res){
42   Customer.update({ userid: 'customerA' },
43      {$set:{cart:req.body.updatedCart}})
44   .exec(function(err, results){
45    if (err || results < 1){
46     res.json(404, {msg: 'Failed to update Cart.'});
47    } else {
48     res.json({msg: "Customer Cart Updated"});
49    }
50   });
51 };
```

Implementing Shopping Cart and Checkout Views

Now that the routes are setup and configured you are ready to implement the views that are rendered by the routes and AngularJS templates. The following sections discuss the main shopping.html view rendered by EJS as well as the various partial views that make up the `cart`, `shipping`, `billing`,`review` and `orders` pages.

Implementing the Shopping View

The shopping view shown in Listing 28.10 is the main view for the shopping application. In fact the user will never actually leave this page, only the content will change using the partial views described in the following sections.

The header of the view registers the `<html ng-app="myApp">` element with the AngularJS `myApp`application and loads the `cart_styles.css` file.
The `<body>` element initializes the AngularJS `ng-controller="shoppingControler"` to provide the functionality to interact with the products, shopping cart, checkout and orders.

The page content changing works using the following `ng-include` directive that maps to`$scope.content`. The content variable in the scope can then be set to which every template file on the server that you want.

```
<div ng-include="content"></div>
```

An example of this is shown in the orders and shopping cart clickable links, shown below, that call `setConent()` with the name of the template file to load.

```
<span class="orders"
  ng-click="setContent('orders.html')">Orders</span>
<span id="cartLink" ng-click="setContent('cart.html')">
 {{customer.cart.length}} items
 <img src="/images/cart.png" />
</span>
```

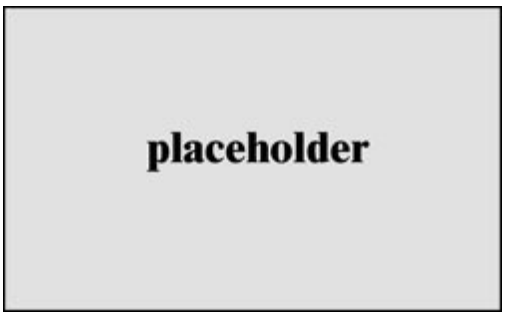
Also notice that the number of items in the cart is taken directly from the scope variable `customer.cart.length`. The customer object in the scope is downloaded directly from the web server when the controller is initialized. Figure 28.1 shows the full rendered shopping.html page.

Listing 28.12. shopping.html Implementing the Main Shopping Page AngularJS Template Code

```
01 <!doctype html>
02 <html ng-app="myApp">
03 <head>
04   <title>Shopping Cart</title>
05   <link rel="stylesheet" type="text/css"
06      href="/static/css/cart_styles.css" />
07 </head>
08 <body>
09   <div ng-controller="shoppingController">
10    <div id="banner">
11     <div id="title">My Store</div>
12     <div id="bar">
13       <span class="orders"
14         ng-click="setContent('orders.html')">Orders</span>
15       <span id="cartLink" ng-click="setContent('cart.html')">
16          {{customer.cart.length}} items
17          <img src="/images/cart.png" />
18       </span>
19     </div>
```

```
20     </div>
21     <div id="main">
22       <div ng-include="content"></div>
23     </div>
24   </div>
25   <script src="/lib/angular/angular.js"></script>
26   <script src="/static/js/cart_app.js"></script>
27 </body>
28 </html>
```

Figure 28.1. Rendered shopping page with the shopping cart link and orders link as well as a list of prints to shop for.



Implementing the Products View

Next we implement the products view to provide the user with a list of products to choose from. In this example the shopping page is very basic only a single page with a list of prints to buy. This is basic, but it is enough to demonstrate the implementation of the shopping cart and it keeps the code simple.

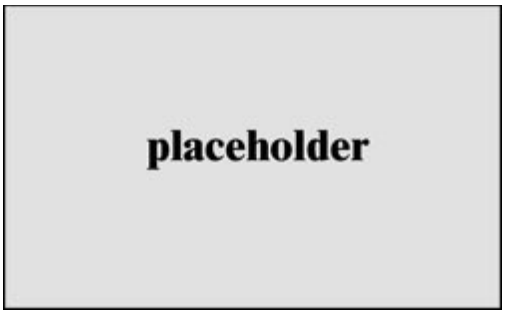
The code in Listing 28.12 is an AngularJS partial that is loaded into the view when the `$scope.content` is set to `products.html`. The code uses an `ng-repeat` on the products that are initialized when the `shoppingController` is initialized. Notice that the product information is displayed on the page using expressions such as `{{product.name}}`.

Also notice that when the user clicks on the `<img>` element that the `setProduct()` function is called in the controller. That function will set the current `$scope.product` value and change the `$scope.content` value to product.html. Figure 28.2 shows the rendered products view.

Listing 28.13. products.html Implementing the Product Listing Template Partial

```
01 <div id="productsContainer">
02   <div class="listItem" ng-repeat="product in products">
03     <img class="listImg" ng-click="setProduct(product._id)"
04         ng-src="../images/{{product.imagefile}}" />
05     <span class="prodName">{{product.name}}</span>
06     <span class="price">{{product.price|currency}}</span>
07   </div>
08 </div>
```

Figure 28.2. Product listing view showing prints available.



Implementing the Product View

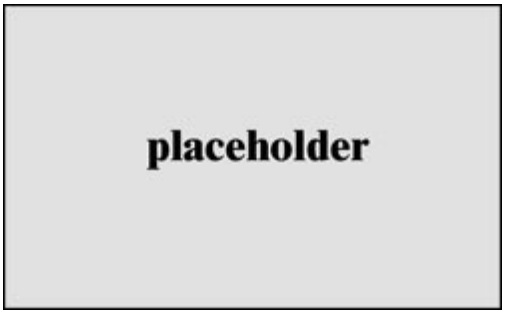
When the user clicks on the image in the products list the product page view is rendered. Listing 28.14shows the AngularJS code used for the product page view. Notice that the product information is displayed using AngularJS expressions that are accessing he `$scope.product` value. The add to cart button sends the `product._id` to the `addToCart()` function in the controller which will add the print to the cart. Figure 28.3 shows the rendered product view that shows
the `image`, `name`,`description`, `quantity` and `price` as well as the add to cart button.

Listing 28.14. product.html Implementing the Product Details Template Partial with Add to Cart Button

```
01 <div id="productsContainer">
02   <div class="listItem" ng-repeat="product in products">
03     <img class="listImg" ng-click="setProduct(product._id)"
04         ng-src="../images/{{product.imagefile}}" />
```

```
05    <span class="prodName">{{product.name}}</span>
06    <span class="price">{{product.price|currency}}</span>
07  </div>
08 </div>
```

Figure 28.3. Product details view showing description, available and add to cart button.



Implementing the Cart View

Once the user clicks on the add to cart button the item is added to the cart and the view is changed to the cart view. The cart view, shown in Listing 28.14, provides a list of products that are currently existing in the cart, a price total and buttons to check out or go back shopping.

The following code implements a delete link in the cart to remove the item. It calls a function `deleteFromCart()` located in the controller an passes the `product._id` to identify which item to delete.

```
<span class="delete"
    ng-click="deleteFromCart(product._id)">Remove</span>
```

Also a quantity field is provided that links directly to the `item.quantity` element where item comes from the ng-repeat of the `customer.shopping` cart array.

The shipping value and total value are calculated by a controller function that is linked by the following expression code. The `|currency` filter is used to format the values of price, shipping and total:

```
<span class="price">{{cartTotal()|currency}}</span>
```

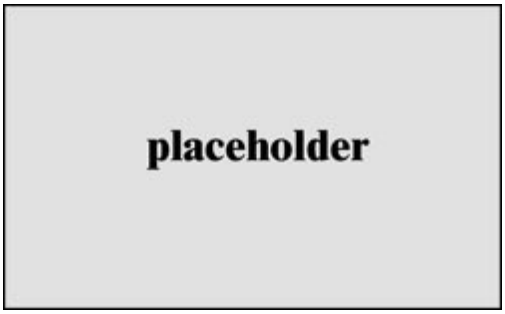
The cool part about AngularJS is that because the `quantity` fields are linked directly to the scope as you change them the shipping and total values change as well. Figure 28.4 shows the rendered shopping cart view.

Listing 28.15. cart.html Implementing the Shopping Cart Template Partial

```
<div id="cartsContainer">
  <div class="listItem" ng-repeat="item in customer.cart"
   ng-init="product=item.product[0]">
   <img class="listImg" ng-click="setProduct(product._id)"
      ng-src="../images/{{product.imagefile}}" />
   <span class="prodName">{{product.name}}</span>
   <span >
    <span class="price">{{product.price|currency}}</span>
    <input class="quantity" type="text" ng-model="item.quantity" />
    <label class="quantity">Quantity</label>
    <span class="delete"
       ng-click="deleteFromCart(product._id)">Remove</span>
  </span>
 </div>
 <hr>
 <div>
  <span>Shipping</span>
  <span class="price">{{shipping|currency}}</span>
 </div>
 <hr>
 <div>
  <span>Total</span>
  <span class="price">{{cartTotal()|currency}}</span>
 </div>
 <hr>
 <div>
   <span class="button" ng-click="checkout()"
      ng-hide="customer.cart.length==0">
    Checkout
   </span>
   <span class="button" ng-click="setContent('products.html')">
    Continue Shopping
```

```
33      </span>
34   </div>
35 </div>
```

Figure 28.4. Shopping cart view allowing the customer to change quantity, remove items and checkout.



Implementing the Shipping View

When the user clicks on the checkout button in the shopping cart they will be directed to the shipping view that allows them to enter shipping information. The shipping view, shown in Listing 28.16, provides a series of input fields to input the shipping information.

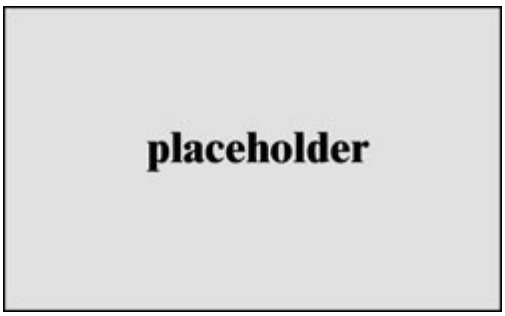
The shipping template code is very straightforward. There are a series of text inputs wit labels for each shipping field value. The fields are linked to the `customer.shipping[0]` object in the scope model using the `ng-model` directive. That way when the user changes the field the scope is automatically changed. You will see that this is very useful when sending customer changes back to the database. When the user clicks on the Continue to Billing button the shipping data will be updated on the server as well and they will be taken to the billing view. The rendered shipping view is shown in Figure 28.5.

Listing 28.16. shipping.html Implementing the Shipping Template Partial

```
01 <div id="shippingContainer">
02   <h2>Ship To:</h2>
03   <label>Name</label>
04   <input type="text" ng-model="customer.shipping[0].name" /><br>
05   <label>Address</label>
06   <input type="text" ng-model="customer.shipping[0].address"
/><br>
```

```
07   <label>City</label>
08   <input type="text" ng-model="customer.shipping[0].city" /><br>
09   <label>State</label>
10   <input type="text" ng-model="customer.shipping[0].state" /><br>
11  <label>Zipcode</label>
12   <input type="text" ng-model="customer.shipping[0].zip" />
13   <hr>
14   <div>
15    <span class="button" ng-click="setShipping()">
16      Continue to Billing
17    </span>
18    <span class="button" ng-click="setContent('products.html')">
19      Continue Shopping
20    </span>
21   </div>
22 </div>
```

Figure 28.5. Shipping view allowing the user to enter in the address to ship the payment to.



Implementing the Billing View

When the user clicks on the Continue to Billing button in the shipping view they will be directed to the billing view that allows them to billing information. The billing view, shown in Listing 28.16, provides a series of input fields to input the billing information.

The billing template code is similar to the shipping template with the addition of a few new fields. The card radio buttons to select the credit card are bound to the `customer.billing[0].cardtype`value in the scope. When you change the radio button selection the model is also changed.

The values for the `<select>` dropdown options come from simple arrays defined in the scope and are bound to the `customer.billing[0]` data as well. For example the following lines use an array of number named months in the scope and binds the `<select>` value to the`customer.billing[0].expiremonth` value in the scope:

```
<select ng-model="customer.billing[0].expiremonth"
      ng-options="m for m in months"></select>
```

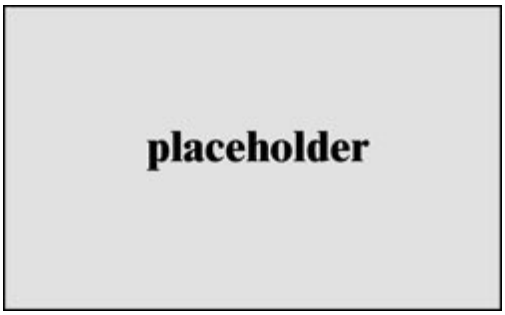
One other thing to note is that the CCV value is passed to the `verifyBilling(ccv)` function when verifying the credit card. The CCV number is not supposed to be stored locally on the customer site so it is kept separate here and passed as its' own parameter. The rendered billing view is shown in Figure 28.6.

Listing 28.17. billing.html Implementing the Billing Template Partial

```
01 <div id="shippingContainer">
02   <h2>Card Info: </h2>
03   <label>Card</label>
04   <input type="radio" ng-model="customer.billing[0].cardtype"
05         value="Visa">  Visa
06   <input type="radio" ng-model="customer.billing[0].cardtype"
07         value="Amex"> Amex
08   <input type="radio" ng-model="customer.billing[0].cardtype"
09         value="MasterCard"> MasterCard
10   <br><label>Name on Card</label>
11  <input type="text" ng-model="customer.billing[0].name" />
12   <br><label>Card Number</label>
13   <input type="text" ng-model="customer.billing[0].number" />
14   <br><label>Expires</label>
15   <select ng-model="customer.billing[0].expiremonth"
16         ng-options="m for m in months"></select>
17   <select ng-model="customer.billing[0].expireyear"
18         ng-options="m for m in years"></select>
19   <label>Card CCV</label>
20   <input class="security" type=text ng-model="ccv" />
21   <h2>Billing Address:</h2>
22   <label>Name</label>
23   <input type="text"
```

```
24        ng-model="customer.billing[0].address[0].name" />
25   <br><label>Address</label>
26   <input type="text"
27        ng-model="customer.billing[0].address[0].address" />
28   <br><label>City</label>
29   <input type="text"
30        ng-model="customer.billing[0].address[0].city" />
31   <br><label>State</label>
32   <input type="text"
33        ng-model="customer.billing[0].address[0].state" />
34   <br><label>Zipcode</label>
35   <input type="text"
36        ng-model="customer.billing[0].address[0].zip" />
37   <hr>
38   <div>
39     <span class="button" ng-click="verifyBilling(ccv)">
40       Verify Billing
41     </span>
42     <span class="button" ng-click="setContent('products.html')">
43       Continue Shopping
44     </span>
45   </div>
46 </div>
```

Figure 28.6. Billing information view allowing the user to enter in a credit card and billing address.

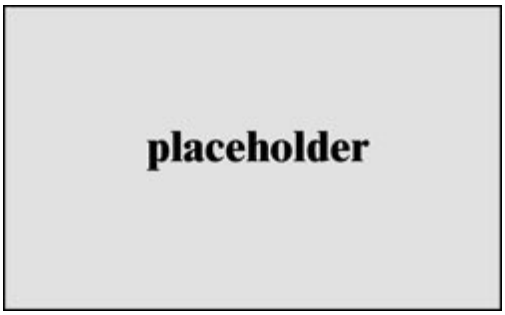


Implementing the Review View

When the user clicks on the Verify Billing button in the billing view they will be taken to the review view so they can review the order along with shipping and billing information. The review view, shown in Listing

28.17, shows the ordered items with totals as well as the shipping and billing information.Figure 28.7 shows the rendered review view.

Figure 28.7. Review view allowing the customer to review their purchase.



Notice that all the information displayed is still coming from the customer object inside the scope. The shipping information comes from `customer.shipping[0]`, the billing information comes from`customer.billing[0]` and the product list comes from `customer.cart`.

When the customer clicks on the Make Purchase button this information will be sent to the webserver and a new order object will be created. The view will then be changed to the order view.

Listing 28.18. review.html Implementing the Order Review Template Partial

```
01 <div id="reviewContainer">
02   <div class="listItem" ng-repeat="item in customer.cart"
03    ng-init="product=item.product[0]">
04    <img class="listImg" ng-click="setProduct(product._id)"
05        ng-src="../images/{{product.imagefile}}" />
06    <span class="prodName">{{product.name}}</span>
07    <span >
08      <span class="price">{{product.price|currency}}</span>
09      <label class="quantity">{{item.quantity}}</label>
10      <label class="quantity">Quantity</label>
11    </span>
12  </div><hr>
13  <div>
14    <span>Shipping</span>
15    <span class="price">{{shipping|currency}}</span>
16  </div><hr>
```

```
<div>
  <span>Total</span>
  <span class="price">{{cartTotal()|currency}}</span>
 </div><hr>
<div>
  <div class="review">
   Shipping:<br>
   {{customer.shipping[0].name}}<br>
   {{customer.shipping[0].address}}<br>
   {{customer.shipping[0].city}},
   {{customer.shipping[0].state}}
   {{customer.shipping[0].zip}}<br>
  </div>
  <div class="review">
   Billing:<br>
   {{customer.billing[0].cardtype}} ending in
   {{customer.billing[0].number.slice(-5,-1)}}<br>
   {{customer.billing[0].address[0].name}}<br>
   {{customer.billing[0].address[0].address}}<br>
   {{customer.billing[0].address[0].city}},
   {{customer.billing[0].address[0].state}}
   {{customer.billing[0].address[0].zip}}<br>
  </div>
 </div>
<div>
  <span class="button" ng-click="makePurchase()">
   Make Purchase
  </span>
  <span class="button" ng-click="setContent('products.html')">
   Continue Shopping
  </span>
 </div>
</div>
```

Implementing the Orders View

When the order is complete the user will be taken to the orders view to see their completed purchases. This is only one method, you could add another page to display the completed order, or you could just send the

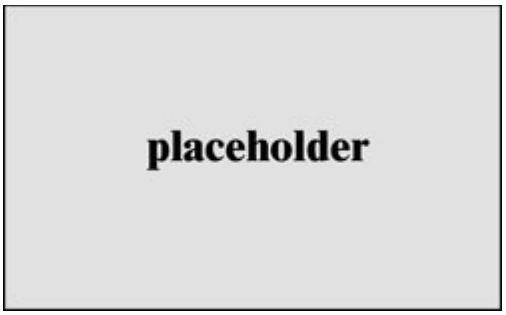
user back to the shopping page. However, for the purpose of just giving an example this method seems as good as any.

The orders view, shown in Listing 28.19, displays a list of orders that this customer has completed. The list comes from an `ng-repeat` directive on the `$scope.orders value`. Within the `ng-repeat`iteration the date placed, status and items bought are listed as shown in Figure 28.8.

Listing 28.19. orders.html Implementing the Orders View Template Partial

```
<div id="reviewContainer">
  <div class="listItem" ng-repeat="order in orders">
    <p class="itemTitle">Order #{{$index+1}}</p>
    <p class="prodDesc">Placed {{order.timestamp|date}}</p>
    <p class="status">{{order.status}}</p>
      <div class="listItem" ng-repeat="item in order.items"
       ng-init="product=item.product[0]">
       <img class="listImg" ng-click="setProduct(product._id)"
           ng-src="../images/{{product.imagefile}}" />
       <span class="prodName">{{product.name}}</span>
       <span >
         <span class="price">{{product.price|currency}}</span>
         <label class="quantity">{{item.quantity}}</label>
         <label class="quantity">Quantity</label>
       </span>
      </div>
  </div>
  <div>
    <span class="button" ng-click="setContent('products.html')">
      Continue Shopping
    </span>
  </div>
</div>
```

Figure 28.8. List of orders that have been placed by this customer.



Adding CSS to Stylize the Views

Listing 28.20 shows the CSS code that is used to style the elements in the shopping cart so that you can see why things look and act the way they do. The CSS is condensed as much as possible to make it fit into the book. Also the titles, buttons and such are implemented large to make it display better in the book's figures.

Listing 28.20. styles.css Implementing the CSS Styles for the View HTML Files

```
01 p{margin:0}
02 label {width:100px; display:inline-block; text-align:right; }
03 input[type="text"]{ border: 2px ridge blue; padding:3px;
04   border-radius:5px; width:400px; }
05 #banner{ border-bottom: 2px blue ridge; height:100px }
06 #title { text-align:center; background-color:#a0d0ff;
07   font:italic bold 48px/60px Georgia, serif; border-radius: 5px }
08 #bar { background-color:#a0d0ff; }
09 #cartLink { float:right; text-align:right; cursor:pointer }
10 #cartLink img { height:25px; }
11 #main {clear:both;}
12 .listItem{border-bottom: 1px solid black; clear:both;
13   margin-top:10px }
14 .listImg { height:50px; vertical-align:top }
15 .fullImg { width:300px; vertical-align:top }
16 .prodName {font: bold 16px/20px Arial, Sans-serif; }
17 .price{ float:right; color:red; width:75px; text-align:right;
18   display:inline-block}
19 .prodInfo{ display:inline-block; }
20 .itemTitle {font: bold 32px/40px Arial, Sans-serif; }
```

```
21 .fullPrice { color:red; font: bold 20px/24px Arial, Sans-serif;
22   text-align:right}
23 .status {color:green; font: bold 14px/18px Arial, Sans-serif;}
24 .prodDesc { font-style: italic; }
25 .button,
26 .cartButton{ font: 18px/24px Arial, Sans-serif; border-radius: 10px;
27   padding:10px; margin-top:35px; cursor: pointer; width:170px;
28   background-image: -webkit-linear-gradient(top, #FFCC66,
#FFFF99);
29   text-align:center}
30 .cartButton img { height:20px; float:right}
31 .button{ display:inline-block; margin:10px;}
32 input.quantity { display:inline-block; float:right; width:30px; }
33 label.quantity { display:inline-block; float:right; width:60px;
34   margin-right:8px; }
35 span.orders,
36 span.delete { cursor:pointer; display:inline-block; float:right;
37   background-color:#FF5858;  border-radius: 8px; text-align:center;
38   font: bold 13px/20px Arial, Sans-serif;
39   margin-right:20px; width:80px; }
40 span.orders{ margin-top:5px; margin-right:10px;
41   background-image: -webkit-linear-gradient(top, #FFCC66,
#FFFF99);}
42 input.security { width:30px }
43 div.review{ display:inline-block; width:45%; vertical-align:top; }
```

Implementing the AngularJS Module and Controller to support Shopping Cart Views

With the views finished you need to implement the AngularJS controller code that will support them. The views need the ability to get the customer, products and orders documents from the webserver. They also need the ability to update the customer shipping, billing and cart as well as the ability to process new orders.

In the shopping cart example everything was built into a single module and controller. The controller code is a bit long for a single section in the book, so the following sections break down the various components of

the controller code so they can be described in smaller code chunks. The full Angular JS code in for the controller is shown in Listing 28.28.

Initializing the Shopping Scope

The first step in implementing the `shoppingController` is to initialize the scope values that we need. The code in Listing 28.21 initializes the shopping scope. The `$scope.months` and`$scope.years` arrays are used to populate the credit card form. The $scope.content determines which AngularJS partial is rendered in the view. It is initialized to `products.html` so the user can begin shopping.

Next there are three $http requests that get the products, customer and orders and use the results to set the `$scope.products`, `$scope.product`, `$scope.customer` and `$scope.orders` objects that are utilized in the AngularJS views.

Listing 28.21. cart_app.js-initialize Implementing the CSS Styles for the View HTML Files

```
004    $scope.months = [1,2,3,4,5,6,7,8,9,10,11,12];
005    $scope.years = [2014,2015,2016,2017,2018,2019,2020];
006    $scope.content = '/static/products.html';
007    $http.get('/products/get')
008     .success(function(data, status, headers, config) {
009       $scope.products = data;
010       $scope.product = data[0];
011     })
012     .error(function(data, status, headers, config) {
013       $scope.products = [];
014     });
015    $http.get('/customers/get')
016     .success(function(data, status, headers, config) {
017      $scope.customer = data;
018     })
019    .error(function(data, status, headers, config) {
020      $scope.customer = [];
021    });
022    $http.get('/orders/get')
023    .success(function(data, status, headers, config) {
```

```
024       $scope.orders = data;
025     })
026     .error(function(data, status, headers, config) {
027       $scope.orders = [];
028     });
```

Implementing Helper Functions

Next we add the helper functions, shown in Listing 28.22, that provide functionality for the AngularJS templates. The `setContent()` function sets the `$scope.content` value effectively changing the view.
The `setProduct()` function is called when a user clicks on a print image and sets the`$scope.product` used in the product view. The `cartTotal()` function will iterate through the products in the users cart and update the `$scope.shipping` and return a total used in the cart and review views.

Listing 28.22. cart_app.js-helpers Implementing the CSS Styles for the View HTML Files

```
029     $scope.setContent = function(filename){
030       $scope.content = '/static/'+ filename;
031     };
032     $scope.setProduct = function(productId){
033       $scope.product = this.product;
034       $scope.content = '/static/product.html';
035     };
036     $scope.cartTotal = function(){
037       var total = 0;
038       for(var i=0; i<$scope.customer.cart.length; i++){
039         var item = $scope.customer.cart[i];
040         total += item.quantity * item.product[0].price;
041       }
042       $scope.shipping = total*.05;
043       return total+$scope.shipping;
044     };
```

Adding Items to the Cart

The `addToCart()` function, shown in Listing 28.23, is called from the template when the user clicks on the add to cart button. The first thing it

does is iterate through the items in the `customer.cart`and if it finds the item is there it increments the quantity otherwise it adds the item to the`customer.cart` array with a quantity of 1.

Once the `$scope.customer` is updated, an `$http POST` is called to the`/customers/update/cart` route to update the cart. That way the cart is persistent and will be there even if you user closes the browser or navigates away. On success the view is switched to `cart.html`. On failure an alert window is displayed.

Listing 28.23. cart_app.js-addToCart styles.css Implementing the CSS Styles for the View HTML Files

```
045    $scope.addToCart = function(productId){
046      var found = false;
047      for(var i=0; i<$scope.customer.cart.length; i++){
048        var item = $scope.customer.cart[i];
049        if (item.product[0]._id == productId){
050          item.quantity += 1;
051          found = true;
052        }
053      }
054      if (!found){
055        $scope.customer.cart.push({quantity: 1,
056                                   product: [this.product]});
057      }
058      $http.post('/customers/update/cart',
059              { updatedCart: $scope.customer.cart })
060       .success(function(data, status, headers, config) {
061         $scope.content = '/static/cart.html';
062       })
063       .error(function(data, status, headers, config) {
064         $window.alert(data);
065       });
066    };
```

Deleting Items from the Cart

The `deleteFromCart()` function, shown in Listing 28.24, is called from the cart template when the user clicks on the remove button. The first thing

it does is iterate through the items in the `customer.cart` and if it finds the item it uses the `array.slice(index,1)` method to delete it from the array.

Once the item is removed from `$scope.customer.cart`, an `$http POST` is called to the `/customers/update/cart` route to update the cart. That way the cart is persistent and will be there even if you user closes the browser or navigates away. On success the view is switched to `cart.html`. On failure an alert window is displayed.

Listing 28.24. cart_app.js-deleteFromCart Implementing the CSS Styles for the View HTML Files

```
067    $scope.deleteFromCart = function(productId){
068      for(var i=0; i<$scope.customer.cart.length; i++){
069        var item = $scope.customer.cart[i];
070        if (item.product[0]._id == productId){
071          $scope.customer.cart.splice(i,1);
072          break;
073        }
074      }
075      $http.post('/customers/update/cart',
076              { updatedCart: $scope.customer.cart })
077       .success(function(data, status, headers, config) {
078         $scope.content = '/static/cart.html';
079       })
080       .error(function(data, status, headers, config) {
081         $window.alert(data);
082       });
083    };
```

Checking Out

The `checkout()` function, shown in Listing 28.25, is called when the user clicks on the checkout button in the cart view. This illustrates how useful AngularJS data binding really is. Since the customer information is always kept up to date, all that is necessary is to send an `$http POST` request with the parameter `{updatedCart:$scope.customer.cart}` to update the cart.

The cart is updated to ensure that any quantity changes made in the cart page will also be persistent later on if the user backs out of the purchase. If the request is successful then the view is switched to `shipping.html`.

Listing 28.25. cart_app.js-checkout Implementing the CSS Styles for the View HTML Files

```
084    $scope.checkout = function(){
085      $http.post('/customers/update/cart',
086              { updatedCart: $scope.customer.cart })
087       .success(function(data, status, headers, config) {
088         $scope.content = '/static/shipping.html';
089       })
090       .error(function(data, status, headers, config) {
091         $window.alert(data);
092       });
093    };
```

Setting Shipping Information

The `setShipping()` function, shown in Listing 28.26, is called when the user clicks on the continue to billing button in the cart view. The shipping information needs to be updated in the database to ensure that it is persistent when the customer leaves the website. An `$http POST` method is called to the `/customers/update/shipping` route. The POST includes the parameter `{updatedShipping:$scope.customer.shipping[0]}` in the body. If the request is successful then the view is switched to `billing.html`. Otherwise an alert is displayed.

Listing 28.26. cart_app.js-setShipping Implementing the CSS Styles for the View HTML Files

```
094    $scope.setShipping = function(){
095      $http.post('/customers/update/shipping',
096         { updatedShipping: $scope.customer.shipping[0] })
097       .success(function(data, status, headers, config) {
098         $scope.content = '/static/billing.html';
099       })
100       .error(function(data, status, headers, config) {
101         $window.alert(data);
```

```
102       });
103    };
```

Verifying Billing

The `verifyBilling()` function, shown in Listing 28.27, is called when the user clicks on the verify billing button in the shipping view. The billing information needs to be updated in the database to ensure that it is persistent when the customer leaves the website. Also the credit card information can be validated at this point on the server. An `$http POST` method is called to the `/customers/update/billing` route. If the request is successful then the view is switched to `review.html`. Otherwise an alert is displayed.

Listing 28.27. cart_app.js-verifyBilling Implementing the CSS Styles for the View HTML Files

```
104    $scope.verifyBilling = function(ccv){
105      $http.post('/customers/update/billing',
106         { updatedBilling: $scope.customer.billing[0], ccv: ccv})
107        .success(function(data, status, headers, config) {
108          $scope.content = '/static/review.html';
109        })
110        .error(function(data, status, headers, config) {
111          $window.alert(data);
112        });
113    };
```

Making the Purchase

The `makePurchase()` function, shown in Listing 28.28, is called when the user clicks on the make purchase button in the billing view. This method sends an `$http POST` method to the `/orders/add` route on the server. The `orderBilling`, `orderShipping` and `orderItems` parameters for the POST request. If the request is successful the `$scope.customer.cart` is initialized to `[]` since the create order code will have done that on customer document in MongoDB.

Also, if the request is successful then a new order document will have been created in the MongoDB database. Therefore another `$http` request

is made this time to the `/orders/get` to get the full list of orders including the new one. Then the view is switched to `orders.html`.

Listing 28.28. cart_app.js-makePurchase Implementing the CSS Styles for the View HTML Files

```
114    $scope.makePurchase = function(){
115      $http.post('/orders/add',
116         { orderBilling: $scope.customer.billing[0],
117           orderShipping: $scope.customer.shipping[0],
118           orderItems: $scope.customer.cart })
119        .success(function(data, status, headers, config) {
120          $scope.customer.cart = [];
121          $http.get('/orders/get')
122          .success(function(data, status, headers, config) {
123            $scope.orders = data;
124            $scope.content = '/static/orders.html';
125          })
126          .error(function(data, status, headers, config) {
127            $scope.orders = [];
128          });
129        })
130        .error(function(data, status, headers, config) {
131          $window.alert(data);
132        });
133    };
```

The full Controller

The code in Listing 28.29 shows the full `myApp` code with the `shoppingController` initialization and all of the controller code together so you can see how thing fit together. Notice that the`shoppingController` definition includes dependencies on `$scope`, `$http` and `$window`. The`$window` dependency allows us to add the browser alert message on processing errors.

Listing 28.29. cart_app.js-full Implementing the CSS Styles for the View HTML Files

```
001 var app = angular.module('myApp', []);
002 app.controller('shoppingController', ['$scope', '$http', '$window',
```

```
003                         function($scope, $http, $window) {
004    $scope.months = [1,2,3,4,5,6,7,8,9,10,11,12];
005    $scope.years = [2014,2015,2016,2017,2018,2019,2020];
006    $scope.content = '/static/products.html';
007    $http.get('/products/get')
008     .success(function(data, status, headers, config) {
009       $scope.products = data;
010       $scope.product = data[0];
011     })
012     .error(function(data, status, headers, config) {
013       $scope.products = [];
014     });
015   $http.get('/customers/get')
016    .success(function(data, status, headers, config) {
017      $scope.customer = data;
018     })
019    .error(function(data, status, headers, config) {
020      $scope.customer = [];
021    });
022    $http.get('/orders/get')
023    .success(function(data, status, headers, config) {
024      $scope.orders = data;
025     })
026     .error(function(data, status, headers, config) {
027      $scope.orders = [];
028     });
029    $scope.setContent = function(filename){
030     $scope.content = '/static/'+ filename;
031   };
032    $scope.setProduct = function(productId){
033     $scope.product = this.product;
034     $scope.content = '/static/product.html';
035   };
036    $scope.cartTotal = function(){
037     var total = 0;
038     for(var i=0; i<$scope.customer.cart.length; i++){
039       var item = $scope.customer.cart[i];
040       total += item.quantity * item.product[0].price;
041     }
```

```
042     $scope.shipping = total*.05;
043     return total+$scope.shipping;
044   };
045   $scope.addToCart = function(productId){
046     var found = false;
047     for(var i=0; i<$scope.customer.cart.length; i++){
048       var item = $scope.customer.cart[i];
049       if (item.product[0]._id == productId){
050         item.quantity += 1;
051         found = true;
052       }
053     }
054     if (!found){
055       $scope.customer.cart.push({quantity: 1,
056                                  product: [this.product]});
057     }
058     $http.post('/customers/update/cart',
059               { updatedCart: $scope.customer.cart })
060      .success(function(data, status, headers, config) {
061        $scope.content = '/static/cart.html';
062      })
063      .error(function(data, status, headers, config) {
064        $window.alert(data);
065      });
066   };
067   $scope.deleteFromCart = function(productId){
068     for(var i=0; i<$scope.customer.cart.length; i++){
069       var item = $scope.customer.cart[i];
070       if (item.product[0]._id == productId){
071         $scope.customer.cart.splice(i,1);
072         break;
073       }
074     }
075     $http.post('/customers/update/cart',
076               { updatedCart: $scope.customer.cart })
077      .success(function(data, status, headers, config) {
078        $scope.content = '/static/cart.html';
079      })
080      .error(function(data, status, headers, config) {
```

```
081         $window.alert(data);
082       });
083     };
084     $scope.checkout = function(){
085       $http.post('/customers/update/cart',
086                 { updatedCart: $scope.customer.cart })
087       .success(function(data, status, headers, config) {
088         $scope.content = '/static/shipping.html';
089       })
090       .error(function(data, status, headers, config) {
091         $window.alert(data);
092       });
093     };
094     $scope.setShipping = function(){
095       $http.post('/customers/update/shipping',
096           { updatedShipping :$scope.customer.shipping[0] })
097       .success(function(data, status, headers, config) {
098           $scope.content = '/static/billing.html';
099       })
100       .error(function(data, status, headers, config) {
101         $window.alert(data);
102       });
103     };
104     $scope.verifyBilling = function(ccv){
105       $http.post('/customers/update/billing',
106           { updatedBilling: $scope.customer.billing[0], ccv: ccv})
107       .success(function(data, status, headers, config) {
108           $scope.content = '/static/review.html';
109       })
110       .error(function(data, status, headers, config) {
111         $window.alert(data);
112       });
113     };
114     $scope.makePurchase = function(){
115       $http.post('/orders/add',
116           { orderBilling: $scope.customer.billing[0],
117             orderShipping: $scope.customer.shipping[0],
118             orderItems: $scope.customer.cart })
119       .success(function(data, status, headers, config) {
```

```
120         $scope.customer.cart = [];
121         $http.get('/orders/get')
122         .success(function(data, status, headers, config) {
123           $scope.orders = data;
124           $scope.content = '/static/orders.html';
125         })
126         .error(function(data, status, headers, config) {
127           $scope.orders = [];
128         });
129       })
130       .error(function(data, status, headers, config) {
131         $window.alert(data);
132       });
133   };
134 }]);
```

Initializing the Application

With the application done you need to create the initial `Customer`, `Order` and `Product` documents in the database. There are several different methods of doing this such as using a database script, creating an admin interface for you application, etc. To make it simple this example includes a basic Node.js script to generate the data that you've seen so far.

The code in Listing 28.30 shows a basic Node.js script that first removes the customers, orders and products collections to clean up if you've already been playing around. It then creates a `Customer`document and an `Order` document and then adds several `Product` documents. The `Product`documents are added to the `Customer` document's cart and the `Order` document's items.

Listing 28.30. cart_init.jsl Initializing the Shopping Cart Application Data in MongoDB

```
01 var mongoose = require('mongoose');
02 var db = mongoose.connect('mongodb://localhost/cart');
03 require('./models/cart_model.js');
04 var Address = mongoose.model('Address');
05 var Billing = mongoose.model('Billing');
```

```
06 var Product = mongoose.model('Product');
07 var ProductQuantity = mongoose.model('ProductQuantity');
08 var Order = mongoose.model('Order');
09 var Customer = mongoose.model('Customer');
10 function addProduct(customer, order, name, imagefile,
11                     price, description, instock){
12   var product = new Product({name:name, imagefile:imagefile,
13                              price:price, description:description,
14                              instock:instock});
15   product.save(function(err, results){
16     order.items.push(new ProductQuantity({quantity: 1,
17                                           product: [product]}));
18     order.save();
19     customer.save();
20      console.log("Product " + name + " Saved.");
21   });
22 }
23 Product.remove().exec(function(){
24    Order.remove().exec(function(){
25      Customer.remove().exec(function(){
26        var shipping = new Address({
27          name: 'Customer A',
28          address: 'Somewhere',
29          city: 'My Town',
30          state: 'CA',
31          zip: '55555'
32        });
33        var billing = new Billing({
34          cardtype: 'Visa',
35          name: 'Customer A',
36          number: '1234567890',
37          expiremonth: 1,
38          expireyear: 2020,
39          address: shipping
40        });
41        var customer = new Customer({
42          userid: 'customerA',
43          shipping: shipping,
44          billing: billing,
```

```
45        cart: []
46      });
47      customer.save(function(err, result){
48        var order = new Order({
49          userid: customer.userid,
50          items: [],
51         shipping: customer.shipping,
52          billing: customer.billing
53        });
54        order.save(function(err, result){
55          addProduct(customer, order, 'Delicate Arch Print',
56              'arch.jpg', 12.34,
57              'View of the breathtaking Delicate Arch in Utah',
58              Math.floor((Math.random()*10)+1));
59          addProduct(customer, order, 'Volcano Print',
60              'volcano.jpg', 45.45,
61              'View of a tropical lake backset by a volcano',
62              Math.floor((Math.random()*10)+1));
63          addProduct(customer, order, 'Tikal Structure Print',
64              'pyramid.jpg', 38.52,
65              'Look at the amazing architecture of early America.',
66              Math.floor((Math.random()*10)+1));
67          addProduct(customer, order, 'Glacial Lake Print',
68              'lake.jpg', 77.45,
69              'Vivid color, crystal clear water from glacial runoff.',
70              Math.floor((Math.random()*10)+1));
71        });
72      });
73    });
74  });
75 });;
```

Summary

In this chapter you got to go through the process of implementing a basic shopping cart using the Node.js, MongoDB and AngularJS web application stack. You were able to define a solid model in Mongoose to support customers, products, orders and the full checkout process.

Also in this chapter you should have recognized the benefits of being able to easily switch between HTML template views in AngularJS and yet have all of your data bound to that page automatically. The data binding value also became very apparent when we were able to update the MongoDB database with only a simple function calls because the `$scope` data was consistently kept up to data as we added items to the cart and change shipping and billing information.

Next

In the next chapter you get a bit of different look as you look at some rich internet application concepts.