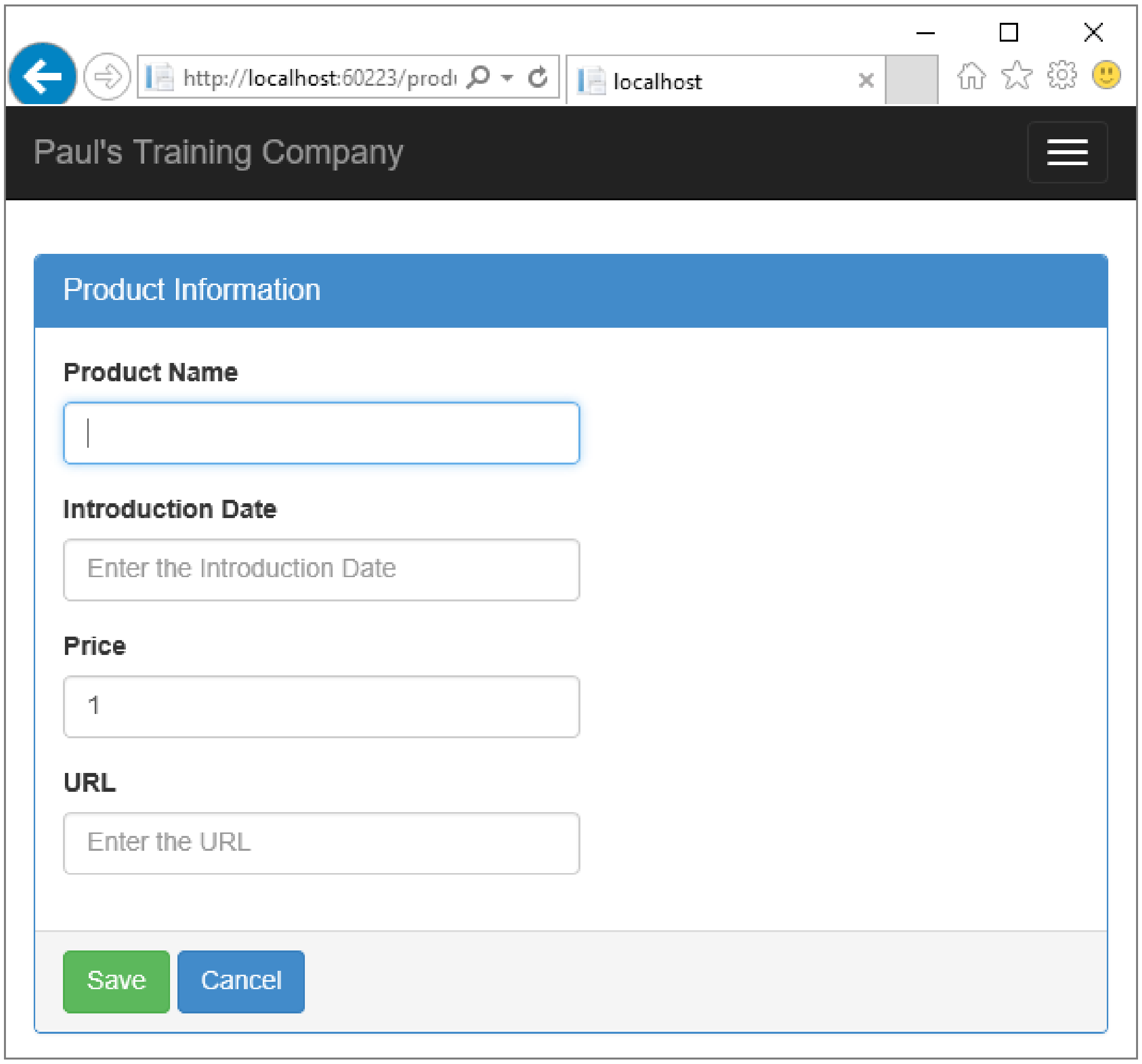
Add Angular to MVC – Part 3

In this blog post, you are going to extend the last sample to allow the user to add a new product. You are going to add a POST method to your Web API controller. You will also create a new Angular component to handle getting and displaying a product record.

Add an Add Button to Product List Page

You need to have a way to get into an “add” mode where you allow a user to enter all the fields to create a new product as shown in Figure 1. You will create the appropriate HTML for this page soon, but first, let’s create an Add button on the main product listing page to get to this page.



Product Information

Product Name

Introduction Date

Enter the Introduction Date

Price

1

URL

Enter the URL

Figure 1: Add new products via a product detail page.

Open the product-list.component.html file, and just below the <h2> tag add the following HTML.

```
<div class="row">
  <div class="col-xs-12">
    <button class="btn btn-primary"
            (click)="add()">
      Add New Product
    </button>
  </div>
</div>
```

When you click on the **Add New Product** button, you want to route to the new product detail page you are going to create. The add() function that is going to be called from the Click event on this button will perform this routing functionality.

Update Product List Component

Let's update the ProductListComponent class to perform this add functionality. Open the product-list.component.ts file and add an import to use the routing service in your class.

```
import { Router } from '@angular/router';
```

Locate the constructor in your ProductListComponent and add a second parameter to this constructor to accept the Router service.

```
constructor(private productService: ProductService,
  private router: Router) {
}
```

Next, add a new function called add(). This function uses the injected router service to call the navigate function. You pass an array to this navigate function. The first parameter is a route to match up with a route you create in your routing component. The second parameter is any parameter you wish to pass. Later you use this product detail page to display an existing product, so you will pass a real product ID as the second parameter. For now, since you are just adding a new product, pass a -1.

```
add() {
  this.router.navigate(['/productDetail', -1]);
}
```

Create Detail HTML

Let's now add the detail page to accept product data from the user. Right mouse click on the \app\product folder and select **Add | HTML page**. Set the name to **product-detail.component.html** and click the OK button. Delete all the HTML in the new page and add the following HTML.

```
<div class="row" *ngIf="product">
  <div class="col-xs-12">
    <div class="panel panel-primary">
      <div class="panel-heading">
        <h1 class="panel-title">Product Information</h1>
      </div>
      <div class="panel-body">
        <!-- ** BEGIN: Error Message Area ** -->
        <div class="row"
             *ngIf="messages && messages.length">
          <div class="col-xs-12">
            <div class="alert alert-warning">
              <ul>
                <li *ngFor="let msg of messages">
                  {{msg}}
                </li>
              </ul>
            </div>
          </div>
        </div>
        <!-- ** END: Error Message Area ** -->

        <!-- ** BEGIN: Detail Entry Area ** -->
        <div class="form-group">
          <label for="productName">Product Name</label>
          <input id="productName"
                 type="text"
                 class="form-control"
                 autofocus="autofocus"
                 placeholder="Enter the Product Name"
                 title="Enter the Product Name"
                 [(ngModel)]="product.productName" />
        </div>
        <div class="form-group">
          <label for="introductionDate">
            Introduction Date</label>
          <input id="introductionDate"
                 type="text"
                 class="form-control"
                 placeholder="Enter the Introduction Date"
                 title="Enter the Introduction Date"
                 [(ngModel)]="product.introductionDate" />
        </div>
        <div class="form-group">
          <label for="price">Price</label>
          <input id="price"
                 type="number"
                 class="form-control"
                 placeholder="Enter the Price"
                 title="Enter the Price"
                 [(ngModel)]="product.price" />
        </div>
        <div class="form-group">
          <label for="url">URL</label>
          <input id="url"
                 type="url"
```

```
                  class="form-control"
                  placeholder="Enter the URL"
                  title="Enter the URL"
                  [(ngModel)]="product.url" />
        </div>
        <!-- ** END: Detail Entry Area ** -->
      </div>
      <div class="panel-footer">
        <button class="btn btn-success"
                (click)="saveProduct()">Save</button>
        <button class="btn btn-primary"
                (click)="goBack()">Cancel</button>
      </div>
    </div>
  </div>
</div>
```

Listing 1: The HTML for the product detail page.

NOTE: I have purposefully left all validation off each input field. You will learn about validation later.

Create Product Detail Component

Now that you have a product detail page, you need a component to go along with it. Right mouse click on the \app\product folder and select **Add | TypeScript file**. Set the name to **product-detail.component.ts**. Add the following code for now. You will add more to this component later.

```
import { Component, OnInit } from "@angular/core";

import { Product } from "./product";

@Component({
  moduleId: module.id,
  templateUrl: "./product-detail.component.html"
})
export class ProductDetailComponent implements OnInit {
  product: Product;
  messages: string[] = [];

  ngOnInit() {
    this.product = new Product();
    this.product.price = 1;
    this.product.url = "http://www.fairwaytech.com";
  }
}
```

Update Routing

You have added the new detail component and HTML. You also wrote code in the ProductListComponent to navigate to this new detail component. However, before you can do that, you need to inform the Angular routing service about the new detail component. Open the app-routing.module.ts file and add a new import statement at the top of this file.

```
import { ProductDetailComponent }
  from "./product/product-detail.component";
```

Add a new route object after the other routes you had previously created. This new route object references the ProductDetailComponent. The *path* property is a little different because you want to pass a parameter name *id* to the ProductDetailComponent class. For the add functionality, you are not going to do anything with this -1 parameter you are passing in, however, for editing, you will pass in a valid product id value in order to retrieve the product record to edit.

```
const routes: Routes = [
  {
    path: 'productList',
    component: ProductListComponent
  },
  {
    path: 'Product/Product',
    redirectTo: 'productList'
  },
  {
    path: 'productDetail/:id',
    component: ProductDetailComponent
  }
];
```

Update AppModule

In the product detail HTML you reference the ngModel directive. However, you have not told your Angular application that you wish to use this directive. In order to do this, open the app.module.ts file and add an import statement for the FormsModule package. This package includes the ngModel directive.

```
import { FormsModule } from '@angular/forms';
```

While you are in this file, also add an import for your new ProductDetailComponent class you added.

```
import { ProductDetailComponent }
  from "./product/product-detail.component";
```

The FormsModule needs to be added to the *imports* property on your NgModule decorator. The ProductDetailComponent should be added to the *declarations* property on your NgModule decorator. Modify the NgModule decorator to look like the following code.

```
@NgModule({
  imports: [BrowserModule, AppRoutingModule,
            HttpModule, FormsModule],
  declarations: [AppComponent,
                 ProductListComponent,
                 ProductDetailComponent],
  bootstrap: [AppComponent],
  providers: [ProductService]
})
```

Run the application, click on the Add New Product button and you should see the detail page appear. Nothing else works at this point, but you just want to verify that you can get to this point.

Add POST Method in Controller

To add the data the user inputs into the product detail page you just created, you are going to need a POST method in your ProductApiController. When you attempt to add a new product, business rules could fail because the user did not fill out the fields correctly. For instance, a product name is required. If the user does not fill one out and the ProductName property gets passed to the server as a blank string, the code generated by the Entity Framework will raise an exception.

The Insert() method in the ProductViewModel handles that situation and converts all the validation errors generated by the Entity Framework into a KeyValuePair<string, string> object. The *key* property in the KeyValuePair object is the name of the property that was in error. The *value* property in the KeyValuePair object is the error message supplied by the Entity Framework. When you call the Insert() method in the ProductViewModel, the Messages property might be filled in with a set of error messages. If this is so, you need to take those messages and pass those back from the Web API.

The mechanism you use to communicate back to the caller that some validation failed on an POST or PUT is to pass back a 400 (BadRequest) and place a ModelStateDictionary object with the set of validation errors as the payload. This means you need to take the list of KeyValuePair objects from the view model and create a ModelStateDictionary object. Open the ProductApiController.cs file and add the following using statement.

```
using System.Web.Http.ModelBinding;
```

One note, on the above using statement. The ModelStateDictionary used by the Web API is different from the one used by MVC controllers. Make sure you are using the ModelStateDictionary class from the above namespace and not the one used by MVC.

Now, you can write the ConvertMessagesToModelState method as shown below.

```
private ModelStateDictionary ConvertMessagesToModelState(
      List<KeyValuePair<string, string>> messages) {
  ModelStateDictionary ret = new ModelStateDictionary();

  foreach (KeyValuePair<string, string> msg in messages) {
    ret.AddModelError(msg.Key, msg.Value);
  }

  return ret;
}
```

Before writing the Post() method, add a using statement so you can use the Product class from the Entity Framework.

```
using PTC.Models;
```

Write the POST method to accept a Product object from the Angular caller. This Post() method simply passes this Product object to the ProductViewModel for processing. You then check the state of the view model to determine what you should return as the IHttpActionResult.

```
[HttpPost]
public IHttpActionResult Post(Product product) {
  IHttpActionResult ret = null;
  ProductViewModel vm = new ProductViewModel();

  if (product != null) {
    if (vm.Insert(product)) {
      ret = Created<Product>(
            Request.RequestUri +
            vm.Entity.ProductId.ToString(),
              vm.Entity);
    }
    else {
      if (vm.Messages.Count > 0) {
        ret = BadRequest(
          ConvertMessagesToModelState(vm.Messages));
      }
      else if (vm.LastException != null) {
        ret = InternalServerError(vm.LastException);
      }
    }
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

Create addProduct Method in Product Service

Now that you have a Web API that can be sent new product data to, let's write code in the ProductService class you created in the earlier blog posts. Open the product.service.ts file and import two new services; Headers and RequestOptions

```
import { Http, Response, Headers, RequestOptions }
  from '@angular/http';
```

Next, add an addProduct() function to this class to post a new product object to the POST method on your ProductApiController class.

```
addProduct(product: Product): Observable<Product> {
  let headers = new Headers({ 'Content-Type':
                              'application/json' });
  let options = new RequestOptions({ headers: headers });

  return this.http.post(this.url, product, options)
    .map(this.extractData)
    .catch(this.handleErrors);
}
```

When you post data, as opposed to getting data, you need to specify the content type as JSON data. You do this by creating a new Headers object and setting the ‘Content-Type’ property to ‘application/json’. Create a RequestOptions object and set the *headers* property to this new Headers object you created. Next, call the post method on the Http service passing in the product object and the RequestOptions object.

Check for Validation Errors

One of three things could happen when you call the Post method. The data will be successfully added to the back-end database table. A set of validation errors is returned via a 400 error. Or, you could get an exception, in which case, a 500 error is sent back. When you wrote the handleErrors() function before, you handled a 404 and a 500 error, but you did not account for a 400. Add a new case statement to handle a 400 in the handleErrors() function.

```
private handleErrors(error: any): Observable<any> {
  let errors: string[] = [];

  switch (error.status)
  {
    case 400:  // Model State Error
      let valErrors = error.json().modelState;
      for (var key in valErrors)
      {
        for (var i = 0; i < valErrors[key].length; i++) {
          errors.push(valErrors[key][i]);
        }
      }
      break;

   ...

   return Observable.throw(errors);
}
```

In this new case statement, you retrieve the *modelState* property and loop through all the key values and retrieve the message from the properties returned. Each of these messages is pushed onto the errors array which is then sent back to the caller via the Observable.throw() function.

Modify Product Detail Component

Now that you have the server-side Web API written, and you created the product service on the client-side to call that, it is not time to add the appropriate code to the ProductDetailComponent class you created earlier to call the product service. First, add three new imports at the top of the product-detail.component.ts file.

```
import { ActivatedRoute, Params } from '@angular/router';
import { Location } from '@angular/common';
import { ProductService } from "./product.service";
```

Add a constructor to this class.

```
constructor(
  private productService: ProductService,
  private route: ActivatedRoute,
  private location: Location
) { }
```

Add a method to allow the user to go back to the previous page if they click on the cancel button.

```
goBack(){
  this.location.back();
}
```

Add a method to your class named handleErrors(). This method is called if the call to the addProduct in the Product Service fails. In this method you loop through the string array of errors and add them to the messages property.

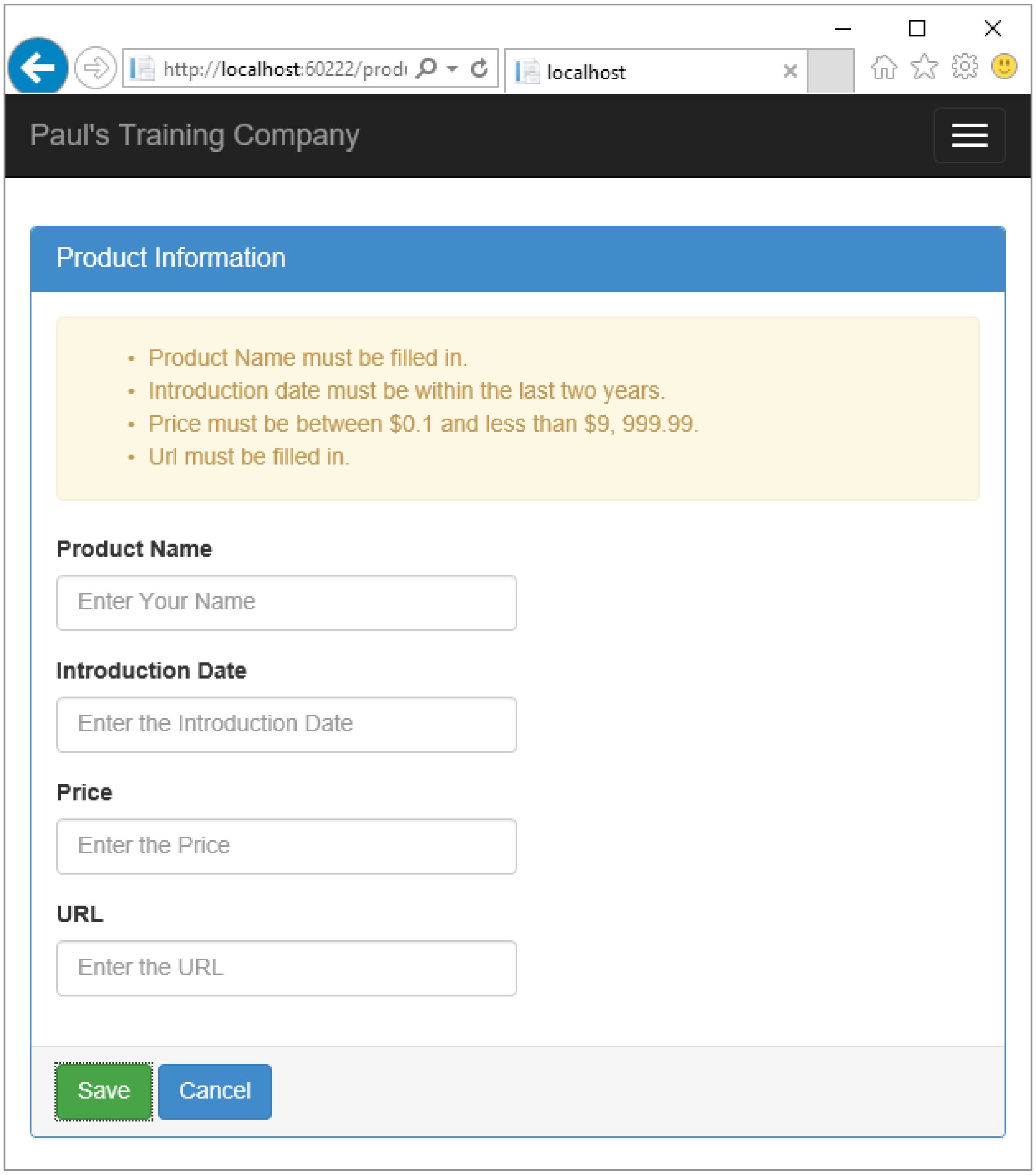
```
private handleErrors(errors: any) {
  for (let msg of errors) {
    this.messages.push(msg);
  }
}
```

Even though you are only performing an add in this blog post, go ahead and add a stub function for updating a product as well. Create three new methods; updateProduct, addProduct and saveProduct().

```
private updateProduct(product: Product) {

}

private addProduct(product: Product) {
  this.productService.addProduct(product)
    .subscribe(() => this.goBack(),
      errors => this.handleErrors(errors));
}

saveProduct() {
  if (this.product) {
    if (this.product.productId) {
      this.updateProduct(this.product);
    }
    else {
      this.addProduct(this.product);
    }
  }
}
```

See the Validation Errors

Run the application, click on the **Add New Product** button. Immediately click on the Save button and you should see a set of validation errors appear on the screen as shown in Figure 2.



- Product Name must be filled in.
- Introduction date must be within the last two years.
- Price must be between $0.1 and less than $9, 999.99.
- Url must be filled in.

Figure 2: Validation errors show up at the top of your detail page.

Add a New Product

Now, go ahead and add some good data for the product, click the Save button and you should be redirected back to the list page where you will see the new product you just added within the list.

Summary

In this blog post you created a detail page to enter new product data. You added a new route to this page and created a component to handle the processing of the new product data. You also added a POST method to your Web API controller. You then created a function in your Angular product service to post the data to the Web API. You also saw how to handle validation errors returned from the server.

Sample Code

You can download the code for this sample at www.pdsa.com/downloads. Choose the category "PDSA Blogs", then locate the sample **Add Angular to MVC – Part 3**. NOTE: After downloading the sample, you will need to right mouse click on the package.json file and select the menu "Restore Packages".