# Add Angular to MVC – Part 4

In this blog post, you will learn to retrieve a single product record using a Web API call from the Angular product service you created. You are going to add Edit and Delete buttons to each row of the HTML table (Figure 1) to allow the user to update and delete an existing product record. For this post, I am assuming you are a Microsoft Visual Studio developer and are familiar with MVC, Angular, C#, and the Web API.

Open up the project you created in the last blog post and follow along with the steps outlined in this one to create the final project.

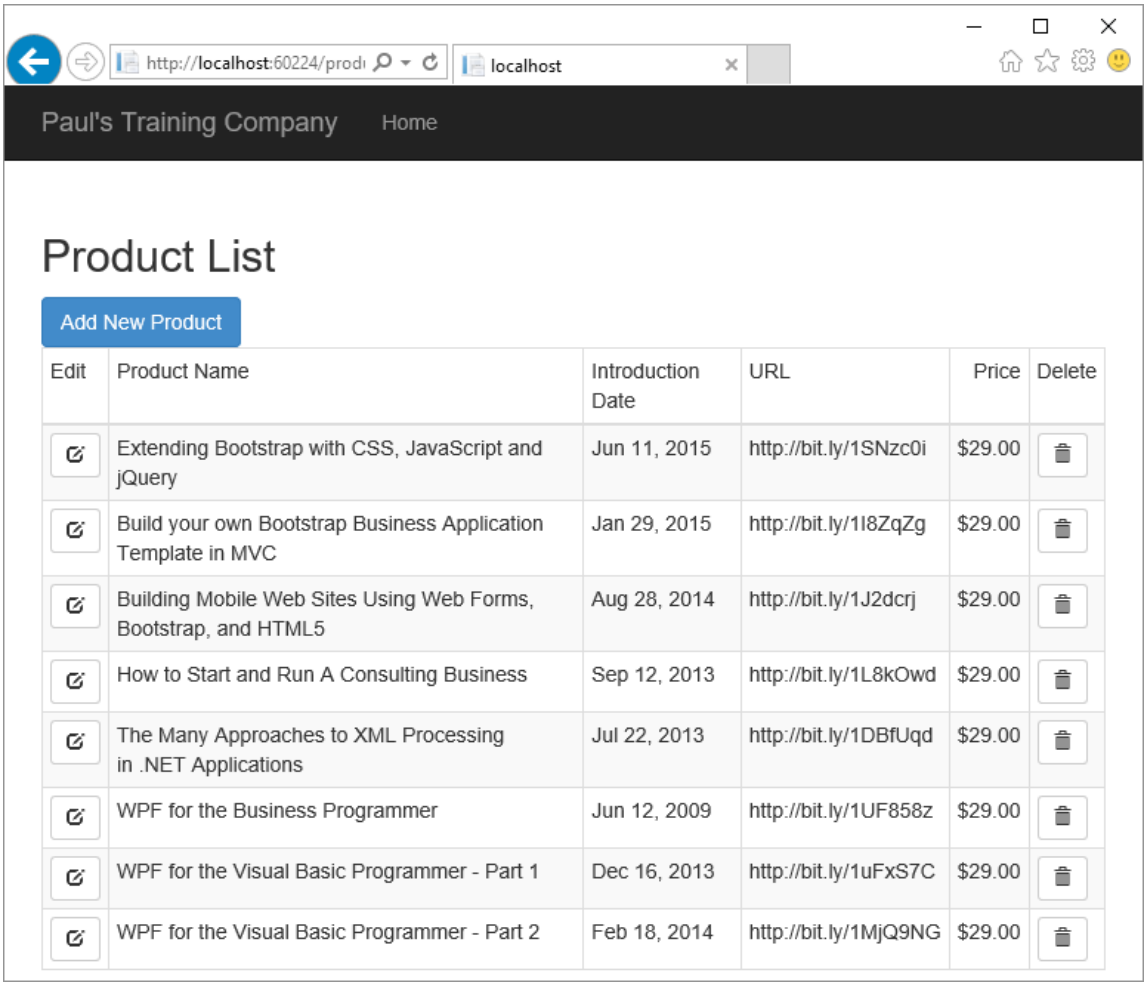


Figure 1: Edit and delete buttons are added to each row in your product table

# Get a Single Product

When the user clicks on the Edit button next to a specific product in the HTML table, you should go back to the server to retrieve the full product record for editing. This ensures you are getting the latest product data.

## Add Get to Controller

To get a single product add a Get() method to your ProductApiController class. This Get() method is different from the other one in this class in that it accepts a product id of the product you wish to retrieve. Add this method, shown below to your ProductApiController class.

```
[HttpGet]
public IHttpActionResult Get(int id) {
  IHttpActionResult ret;
  ProductViewModel vm = new ProductViewModel();

  vm.GetProduct(id);
  if (vm.Entity != null) {
    ret = Ok(vm.Entity);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

## Add Get to Angular Product Service

Now that you have the Web API method created, write a getProduct() method in the ProductService component. Open the product.service.ts file and add the following method.

```
getProduct(id: number): Observable<Product> {
  let url = this.url + "/" + id;
  return this.http.get(url)
    .map(response => response.json() as Product)
    .catch(this.handleErrors);
}
```

This method builds a URL that looks like the following: **api/productApi/2**. The number 2 on the end is what gets passed to the *id* parameter in the Get() method in your ProductApiController.

# Add Select Button to HTML Table

As you saw in Figure 1, you need to add an “Edit” column to your table. Open the product-list.component.html file and insert a new <td> element within the <thead> element.

```
<td>Edit</td>
```

Move down to the <tbody> element and insert a new <td> element in the same position.

```
<td>
  <button class="btn btn-default btn-sm"
          (click)="selectProduct(product.productId)">
    <i class="glyphicon glyphicon-edit"></i>
  </button>
</td>
```

# Modify Product List Component

The click event on this button is going to call a method named selectProduct(). You pass in the product id to this method in order to pass this id to the detail page so it can load the product data associated with that id. Add the selectProduct() function to the ProductListComponent. This function is going to call the navigate function and pass that id to the product detail page.

```
selectProduct(id: number) {
  this.router.navigate(['/productDetail', id]);
}
```

# Retrieve a Passed Parameter

You are going to need to modify the ngOnInit() method you previously wrote in the ProductDetailComponent. When you created the add functionality in the last blog post, you did not do anything with the parameter that was passed into the ProductDetailComponent. Now, since you are passing an id, you are going to use that id to call the getProduct method on the product service to retrieve the product. Open the product-detail.component.ts file and modify the ngOnInit() function to look like the following.

```
ngOnInit() {
  this.route.params.forEach((params: Params) => {
    if (params['id'] !== undefined) {
      if (params['id'] != "-1") {
        this.productService.getProduct(params['id'])
          .subscribe(product => this.product = product,
              errors => this.handleErrors(errors));
      }
      else {
        this.product = new Product();
        this.product.price = 1;
        this.product.url = "http://www.fairwaytech.com";
      }
    }
  });
}
```

In the above code you loop through the route.params array and retrieve a Params object. You check to see if the 'id' parameter is defined on that Params object. If the id value exists, check if that value to see if it is equal to a -1. If so, then assume you are adding a product. If the value is anything else, then it is a valid product id. Call the getProduct() method on the product service to retrieve a single product object.

At this point you can run the application and click on the Edit button. If you did everything correctly, you should see product data in all the input fields.

# Update a Product

Now that you have the current product data displayed in the input fields, the user may update them. Create the appropriate code to perform the updating now.

## Add PUT Method in Controller

Our first step, of course, is to add a new PUT method in the ProductApiController class. Open the ProductApiController.cs file and add the following code.

```
[HttpPut()]
public IHttpActionResult Put(int id, Product product) {
  IHttpActionResult ret = null;
  ProductViewModel vm = new ProductViewModel();

  if (product != null) {
    if (vm.Update(product)) {
      ret = Ok(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;
}
```

# Add updateProduct to Product Service

Add another method to your Angular product service class to call this new PUT method. Open the product.service.ts file and add the following updateProduct() method.

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

  return this.http.put(this.url + "/" + product.productId,
                       product, options)
    .map(this.extractData)
    .catch(this.handleErrors);
}
```

When you PUT 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. Call the put method on the Http service passing in the product object and the RequestOptions object. When calling a PUT you specify the URL and you also add on the product id to that URL.

## Modify the updateProduct() Method

In the last blog post you wrote an updateProduct method that was empty. It is now time to fill in this method. Open the product-detail.component.ts file and modify the updateProduct method to call the updateProduct method you created in the product service.

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

# Delete a Product

The last piece of functionality to add to your product page is the ability to delete product.

## Add DELETE to Controller

Add a DELETE method to your ProductApiController to which you pass in the product id to delete. Open the ProductApiController.cs file and add the Delete method shown below.

```
[HttpDelete]
public IHttpActionResult Delete(int id) {
  IHttpActionResult ret;
  ProductViewModel vm = new ProductViewModel();

  if (vm.Delete(id)) {
    ret = Ok(vm.Entity);
  }
  else {
    ret = NotFound();
  }

  return ret;
}
```

# Add deleteProduct to ProductService

Add a method to your Angular product service to call the delete method in your Web API. Open the product.service.ts file and add the deleteProduct() method shown below.

```
deleteProduct(id: number): Observable<Product> {
  return this.http.delete(this.url + "/" + id)
    .map(() => null)
    .catch(this.handleErrors);
}
```

# Add Delete Button to HTML Table

As you saw in Figure 1, you need to add a “Delete” column to your table. Open the product-list.component.html file and insert a new <td> element within the <thead> element. Make this the last element in the <thead> element.

```
<td>Delete</td>
```

Add a <td> as the very last element in the <tbody> tag. Add a button with a click event that calls a method in your ProductListComponent class. Pass the current product id in the table to this method.

```
<td>
  <button class="btn btn-default btn-sm"
          (click)="deleteProduct(product.productId)">
    <i class="glyphicon glyphicon-trash"></i>
  </button>
</td>
```

# Add deleteProduct() Method in List Component

Now that you have a button to call a deleteProduct method, go ahead and add that method. Open the product-list.component.ts file and add the code shown below.

```
deleteProduct(id: number) {
  if (confirm("Delete this product?")) {
    this.productService.deleteProduct(id)
      .subscribe(() => this.getProducts(),
      errors => this.handleErrors(errors));
  }
}
```

This method first confirms with the user that they really wish to delete this product. If they respond affirmatively, the deleteProduct() method on the Angular product service is called. If the delete is successful, the getProducts method is called to refresh the collection of products from the server and redisplay the list of products.

# Summary

In this blog post you finished your product page by learning to retrieve, update and delete an existing product record. In these four blog posts you learned how to replace a single MVC page with an Angular page. There is no need to completely rewrite an MVC application. Instead, you can just replace a few pages that might need the performance of Angular.

# 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 4**. NOTE: After downloading the sample, you will need to right mouse click on the package.json file and select the menu “Restore Packages”. You also need to create the Product table in a SQL Server database and update the connection string to point to your server and database name.