.NET 10 App Dev Hands-On Lab

MVC Lab 6 –Controllers and Views

This lab walks you through creating the `BaseCrudController` and finishing the `CarsController`. Once the controllers are completed, the application's views are added and/or updated. Before starting this lab, you must have completed MVC Lab 5.

Note: Adjust any directory separators to your OS (e.g. \ for Windows, / for Mac/Linux).

Part 1: Create the BaseCrudController

Copilot Agent Mode

The following prompts will complete Steps 1-6. Please verify that the generated code matches the lab document.

```
Setup Prompt: Always use file scoped namespaces. Always combine attributes on a single line when
possible. The project does not use nullable reference types. There is a GlobalUsings.cs file that
includes common usings, don't include using statements in new files if they are already in the
globalusings.cs file. I prefer expression bodied members when possible. Single line if statements
should still use braces. Use ternary operators when appropriate. Use internal over private. All
classes and methods are public unless told otherwise. Use default primary constructors when
possible and don't declare a class level variable if the parameter from the constructor can be
used. All work is to be done in the AutoLot.Mvc project unless otherwise specified.

Prompt: Add the following global usings to the GlobalUsings.cs file if they do not already exist
(sorted alphabetically. Don't remove any existing global using statements).
global using AutoLot.Dal.Repos.Base;
global using AutoLot.Dal.Repos.Interfaces.Base;
global using AutoLot.Models.Entities.Base;
global using Microsoft.AspNetCore.Mvc.Rendering;
```

```
Prompt: In the Controllers folder, add a new folder named Base and add a new public abstract class
named BaseCrudController. Make it generic with TEntity, constrained to BaseEntity, new(). Add the
Route (controller/action) to the top of the class. Accept two parameters in the , IAppLogging
(appLogging) and IBaseRepo<TEntity> (baseRepo). Assign the parameters to two protected readonly
properties named AppLoggingInstance and BaseRepoInstance, respectively. Add the following
protected methods:
abstract GetLookupValues (input:none. return type: SelectList)
GetOneEntity (input:int? id. return type: TEntity. Logic:return null if id is null, else call Find
on BaseRepoInstance)
Add the following public virtial action methods:
HttpGet Index - add Route(/Controller) and Route(/Controller/Action) attributes. (input:none.
return type: IActionResult. Logic: return View with call to GetAllIgnoreQueryFilters as the model)
HttpGet("{id?}") Details -  (input int?. return IActionResult. Logic: if id does not have a value,
return BadRequest. Else call GetOneEntity. If Entity is null, return NotFound(). else return
View(entity)
HttpGet Create Assign ViewData["LookupValues] = GetLookupValues(), return View
HttpPost Create (input TEntity entity. return IActionResult. Logic: if ModelState is not valid,
assign ViewData["LookupValues"] = GetLookupValues(), return View(entity). Else call Add on
BaseRepoInstance with entity, then redirect to Details)
HttpGet("{id?}) Edit (input int? id. return IActionResult. Logic: If id.hasvalue isn't true,
return BadRequest(). If GetOneEntity returns null, return NotFound(). Assign
ViewData["LookupValues] = GetLookupValues(), return View(entity)
HttpPost("{id}") Post (input int id, TEntity entity. return IActionResult. Logic: if id !=
entity.Id, return BadRequest, if ModelState is not valid, assign ViewData["LookupValues"] =
GetLookupValues(), return View(entity). Else call Update on BaseRepoInstance with entity, then
redirect to Details)
HttpGet("{id?}) Delete (input int? id. return IActionResult. Logic: If id.hasvalue isn't true,
return BadRequest(). If GetOneEntity returns null, return NotFound(). return View(entity)
HttpPost("{id}") Post (input int id, TEntity entity. return IActionResult. Logic: if id !=
entity.Id, return BadRequest, Call Delete on BaseRepoInstance with entity, then redirect to Index)

Prompt: Add the following global usings to the GlobalUsings.cs file if they do not already exist
(sorted alphabetically. Don't remove any existing global using statements).
global using AutoLot.Mvc.Controllers.Base;
```

Manual

Complete Steps 1-6, then proceed to Part 2.

Step 1: Update the global using statements

- Add the following global using statements to the `GlobalUsings.cs` file in the `AutoLot.Mvc` project:

```
global using AutoLot.Dal.Repos.Base;
global using AutoLot.Dal.Repos.Interfaces.Base;
global using AutoLot.Models.Entities.Base;
global using Microsoft.AspNetCore.Mvc.Rendering;
```

Step 2: Create the BaseCrudController class, constructor, and helper methods

- Create a new folder named `Base` in the `Controllers` folder, and in this folder, create a new class named `BaseCrudController`. Make the class `public abstract` and inherit `Controller`. Make it generic, taking in an entity type. In the default constructor, pass in the logging and repo dependencies. Finally, add the default route to the controller:

```
namespace AutoLot.Mvc.Controllers.Base;

[Route("[controller]/[action]")]
public abstract class BaseCrudController<TEntity>(
    IAppLogging appLogging,
    IBaseRepo<TEntity> baseRepo) : Controller where TEntity : BaseEntity, new()
{
  protected readonly IAppLogging AppLoggingInstance = appLogging;
  protected readonly IBaseRepo<TEntity> BaseRepoInstance = baseRepo;
  //rest of the code goes here
}
```

- Add an abstract function that returns a `SelectList` of look-up values (like Makes) and a helper function to get a single entity:

```
protected abstract SelectList GetLookupValues();
protected TEntity GetOneEntity(int? id) => id == null ? null : BaseRepoInstance.Find(id.Value);
```

Step 3: Add the Index and Details action methods

- Create the `Index` and `Details` action methods, set the routing, and return all entities:

```
[Route("/[controller]")]
[Route("/[controller]/[action]")]
[HttpGet]
public virtual IActionResult Index() => View(BaseRepoInstance.GetAllIgnoreQueryFilters());

[HttpGet("{id?}")]
public virtual IActionResult Details(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest();
  }
  var entity = await GetOneEntityAsync(id);
  return entity == null ? NotFound() : View(entity);
}
```

Step 4: Add the Create Action Methods

- Add the `HttpGet and HttpPost Create` Action Methods:

```
[HttpGet]
public virtual IActionResult Create()
{
  ViewData["LookupValues"] = GetLookupValues();
  return View();
}
[HttpPost][ValidateAntiForgeryToken]
public virtual IActionResult Create(TEntity entity)
{
  if (!ModelState.IsValid)
  {
    ViewData["LookupValues"] = GetLookupValues();
    return View(entity);
  }
  BaseRepoInstance.Add(entity);
  return RedirectToAction(nameof(Details), new { id = entity.Id });
}
```

Step 5: Add the Edit Action Methods

- Add the `HttpGet` and `HttpPost Edit` Action Methods:

```
[HttpGet("{id?}")]
public virtual IActionResult Edit(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest();
  }
  var entity = GetOneEntity(id);
  if (entity == null)
  {
    return NotFound();
  }
  ViewData["LookupValues"] = GetLookupValues();
  return View(entity);
}
[HttpPost("{id}")][ValidateAntiForgeryToken]
public virtual IActionResult Edit(int id, TEntity entity)
{
  if (id != entity.Id)
  {
    return BadRequest();
  }
  if (!ModelState.IsValid)
  {
    ViewData["LookupValues"] = GetLookupValues();
    return View(entity);
  }
  BaseRepoInstance.Update(entity);
  return RedirectToAction(nameof(Details), new { id = entity.Id });
}
```

Step 6: Add the Delete Action Methods

- Add the `HttpGet` and `HttpPost Delete` Action Method:

```
[HttpGet("{id?}")]
public virtual IActionResult Delete(int? id)
{
  if (!id.HasValue)
  {
    return BadRequest();
  }
  var entity = GetOneEntity(id);
  return entity == null ? NotFound() : View(entity);
}
[HttpPost("{id}")][ValidateAntiForgeryToken]
public virtual IActionResult Delete(int id, TEntity entity)
{
  if (id != entity.Id)
  {
    return BadRequest();
  }
  BaseRepoInstance.Delete(entity);
  return RedirectToAction(nameof(Index));
}
```

- Add the following global using statements to the `GlobalUsings.cs` file:

```
global using AutoLot.Mvc.Controllers.Base;
```

Part 2: Update the Cars Controller

Copilot Agent Mode

The following prompts will complete Steps 1-3. Please verify that the generated code matches the lab document.

```
Prompt: Remove the route from the CarsController and have it inherit from BaseCrudController with
TEntity as Car. Use a primary contructor to accept IAppLogging and ICarRepo parameters and pass
them to the base constructor. Also take in a IMakeRepo (makeRepo) parameter (do not assign it to a
class field. Override the GetLookupValues method to return a SelectList of all Makes from the
makeRepo ordered by Name, with Id as the value field and Name as the text field. Add another
action method:
HttpGet("{makeId}/{mAKEnAME}") ByMake (input int makeId, string makeName. return IActionResult.
Logic: return View with a model of the result of casting BaseRepoInstance to ICarRepo and calling
GetAllBy passing in the makeId. Set the ViewBag.MakeName = makeName.
(no HTTP verb) BadEndPoint (input:none. return IActionResult. Logic: return new OkObjectResult(5))
Comment out this method after adding it.
```

Manual

Complete Steps 1-3, then proceed to Part 3.

Step 1: Update the class to inherit from the BaseCrudController and Implement the SelectList Helper Function

- Remove the route (it comes from the base class) and inherit from `BaseCrudController`. Next, **delete all the action methods** and add a primary constructor that takes instances of `IAppLogging<T>`, `ICarRepo`, and `IMakeRepo`:

```
namespace AutoLot.Mvc.Controllers;

public class CarsController(IAppLogging appLogging, ICarRepo baseRepo, IMakeRepo makeRepo)
  : BaseCrudController<Car>(appLogging, baseRepo)
{
  //implementation goes here
}
```

- Override the abstract function to get the `SelectList` from the Makes:

```
protected override SelectList GetLookupValues()
  => new SelectList(makeRepo.GetAll().OrderBy(m => m.Name), nameof(Make.Id), nameof(Make.Name));
```

Step 2: Add the ByMake Action Method

- Update the `ByMake` action method, set the routing, and return all cars for a certain make:

```
[HttpGet("{makeId}/{makeName}")]
public IActionResult ByMake(int makeId, string makeName)
{
  ViewBag.MakeName = makeName;
  return View(((ICarRepo)BaseRepoInstance).GetAllBy(makeId));
}
```

Step 3: Add the BadEndpoint ActionMethod

- The `BadEndpoint` action method doesn't have a verb specified, which makes it a security risk:

```
public IActionResult BadEndPoint() => new OkObjectResult(5);
```

- You can hit this with any verb (e.g. using Bruno). When done testing, either comment out this code or add the `HttpGet` verb:

```
[HttpGet]
public IActionResult BadEndPoint() => new OkObjectResult(5);
```

Part 3: General Views

Step 1: Add the SimpleService View

- The `HomeController` has two methods that demonstrate using keyed dependencies. This view will service both methods. Add a new view named `SimpleService.cshtml` in the `Views\Home` folder and update it to the following:

```
@model string
<h1>@Model</h1>
```

- Add the following to the `_Menu.cshtml` partial view:

```
<li class="nav-item dropdown">
  <a class="nav-link dropdown-toggle text-dark" data-bs-toggle="dropdown">
    DI <i class="fa fa-syringe"></i>
  </a>
  <div class="dropdown-menu">
    <a class="dropdown-item text-dark" asp-controller="Home" asp-action="GetServiceOne">
      Service One</a>
    <a class="dropdown-item text-dark" asp-controller="Home" asp-action="GetServiceTwo">
      Service Two</a>
  </div>
</li>
```

Step 2: The RazorSyntax Action Method and View

- Add a new action method named `RazorSyntax` in the `HomeController`:

```
[HttpGet]
public IActionResult RazorSyntax([FromServices] ICarRepo carRepo)
{
  var car = carRepo.Find(1);
  return View(car);
}
```

- Update the `_Menu.cshtml` partial view to include a menu item for the new view:

```
<li class="nav-item">
  <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="RazorSyntax">Razor
Syntax <i class="fas fa-cut"></i></a>
</li>
```

- Create a new view named `RazorSyntax` in the `Views\Home` directory and update it to the following:

```
@model Car
@{
  ViewData["Title"] = "Razor Syntax";
}

<h1>Razor Syntax</h1>

@for (int i = 0; i < 15; i++)
{
  //do something
}
@{
    //Code Block
    var foo = "Foo";
    var bar = "Bar";
    var htmlString = "<ul><li>one</li><li>two</li></ul>";
}
@foo<br />
@htmlString<br />
@foo.@bar<br />
@foo.ToUpper()<br/>
@Html.Raw(htmlString)
<hr />
@{
    @:Straight Text
    <div>Value:@Model.Id</div>
    <text>
        Lines without HTML tag
    </text>
    <br />
}

<hr/>
@*
    Multiline Comments
    Hi.
*@
Email Address Handling:<br/>
foo@foo.com = foo@foo.com<br/>
@@foo<br/>
test@foo = test@foo<br/>
test@(foo) = testFoo<br/>
<hr/>
@functions {
    public static IList<string> SortList(IList<string> strings)  {
        var list = from s in strings orderby s select s;
        return list.ToList();
    }
}
-------------------------------------------------------------
@{
    var myList = new List<string> {"C", "A", "Z", "F"};
    var sortedList = SortList(myList); //MyFunctions.SortList(myList)
}
```

```
@foreach (string s in sortedList)
{
    @s@:&nbsp;
}
<hr/>
@{
    Func<dynamic, object> b = @<strong>@item</strong>;
}
This will be bold: @b("Foo")<hr/>
<a asp-controller="Cars" asp-action="Details" asp-route-id="@Model.Id">@Model.PetName</a>
```

Step 3: Update the Index View

- Update the `Home/Index.cshtml` view to use the `DealerInfo` passed in from the action method and inject the Keyed Service. Update the view to the following:

```
@model AutoLot.Services.ViewModels.DealerInfo
@inject IServiceProvider serviceProvider
@{
  ViewData["Title"] = "Home Page";
  var service = serviceProvider.GetKeyedService<ISimpleService>(nameof(SimpleServiceOne));
}
<div class="text-center">
  <h1 class="display-4">Welcome to @Model.DealerName</h1>
  <p class="lead">Located in @Model.City, @Model.State</p>
</div>
<div>
  @if (service != null)
  {
    <p>@service.SayHello()</p>
  }
</div>
```

Part 4: The Car Views

Step 1: The Partial and Template Views

- Create a new folder named `Cars` under the `Views` folder. In this folder, create three new folders, `DisplayTemplates`, `EditorTemplates`, and `Partials`.
- Add a new view named `Car.cshtml` under the `Views\Cars\DisplayTemplates` folder. Update the markup to the following:

```
@model Car
<dl class="row">
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.MakeId)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.MakeNavigation.Name)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.Color)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.Color)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.PetName)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.PetName)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.Price)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.Price)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.DateBuilt)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.DateBuilt)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.IsDrivable)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.IsDrivable)</dd>
</dl>
```

- Add a new view named `CarWithColors.cshtml` under the `Views\Cars\DisplayTemplates` folder. Update the markup to the following:

```
@model Car
<hr />
<dl class="row">
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.MakeId)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.MakeNavigation.Name)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.Color)</dt>
  <dd class="col-sm-10" style="color:@Model.Color">@Html.DisplayFor(model => model.Color)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.PetName)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.PetName)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.Price)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.Price)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.DateBuilt)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.DateBuilt)</dd>
  <dt class="col-sm-2">@Html.DisplayNameFor(model => model.IsDrivable)</dt>
  <dd class="col-sm-10">@Html.DisplayFor(model => model.IsDrivable)</dd>
</dl>
```

- Add a new view named `Car.cshtml` under the `Views\Cars\EditorTemplates` folder. Update the markup to the following:

```
@model Car
<div asp-validation-summary="All" class="text-danger"></div>
<div>
  <label asp-for="MakeId" class="col-form-label"></label>
  <select asp-for="MakeId" class="form-control" asp-items="@ViewBag.LookupValues"></select>
  <span asp-validation-for="MakeId" class="text-danger"></span>
</div>
<div>
  <label asp-for="Color" class="col-form-label"></label>
  <input asp-for="Color" class="form-control"/>
  <span asp-validation-for="Color" class="text-danger"></span>
</div>
<div>
  <label asp-for="PetName" class="col-form-label"></label>
  <input asp-for="PetName" class="form-control" />
  <span asp-validation-for="PetName" class="text-danger"></span>
</div>
<div>
  <label asp-for="Price" class="col-form-label"></label>
  <input asp-for="Price" class="form-control"/>
  <span asp-validation-for="Price" class="text-danger"></span>
</div>
<div>
  <label asp-for="DateBuilt" class="col-form-label"></label>
  <input asp-for="DateBuilt" class="form-control"/>
  <span asp-validation-for="DateBuilt" class="text-danger"></span>
</div>
<div>
  <label asp-for="IsDrivable" class="col-form-label"></label>
  <input asp-for="IsDrivable" />
  <span asp-validation-for="IsDrivable" class="text-danger"></span>
</div>
```

- Add a new view named `_CarList.cshtml` under the `Views\Cars\Partials` folder. Update the markup to the following:

```
@model IEnumerable<Car>
@{
  var showMake = true;
  if (bool.TryParse(ViewBag.ByMake?.ToString(), out bool byMake))
  {
    showMake = !byMake;
  }
}
<p><item-create></item-create></p>
<table class="table">
  <thead>
    <tr>
      @if (showMake)
      {
        <th>@Html.DisplayNameFor(model => model.MakeId) </th>
      }
      <th>@Html.DisplayNameFor(model => model.Color)</th>
      <th>@Html.DisplayNameFor(model => model.PetName)</th>
      <th>@Html.DisplayNameFor(model => model.Price)</th>
      <th>@Html.DisplayNameFor(model => model.DateBuilt)</th>
      <th>@Html.DisplayNameFor(model => model.IsDrivable)</th>
      <th></th>
    </tr>
  </thead>
  <tbody>
  @foreach (var item in Model)
  {
    <tr>
      @if (showMake)
      {
        <td>@Html.DisplayFor(modelItem => item.MakeNavigation.Name)</td>
      }
      <td>@Html.DisplayFor(modelItem => item.Color)</td>
      <td>@Html.DisplayFor(modelItem => item.PetName)</td>
      <td>@Html.DisplayFor(modelItem => item.Price)</td>
      <td>@Html.DisplayFor(modelItem => item.DateBuilt)</td>
      <td>@Html.DisplayFor(modelItem => item.IsDrivable)</td>
      <td>
        <item-edit item-id="@item.Id"></item-edit> |
        <item-details item-id="@item.Id"></item-details> |
        <item-delete item-id="@item.Id"></item-delete>
      </td>
    </tr>
  }
  </tbody>
</table>
```

Step 2: Update the RazorSyntax View

- Update the bottom of the RazorSyntax view to match the following:

```
This will be bold: @b("Foo")
<hr />
@* If the templates were in the shared folder, you wouldn't need the full path listed*@
@Html.DisplayForModel("../Cars/DisplayTemplates/Car.cshtml")
@Html.DisplayForModel("../Cars/DisplayTemplates/CarWithColors.cshtml")
<hr/>
@Html.EditorForModel("../Cars/EditorTemplates/Car.cshtml")
<hr/>
<a asp-controller="Cars" asp-action="Details" asp-route-id="@Model.Id">@Model.PetName</a>
```

Step 3: Create the Index and ByMake views

- Add a new view named `Index.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model IEnumerable<Car>
@{
  ViewData["Title"] = "Index";
}
<h1>Vehicle Inventory</h1>
<partial name="Partials/_CarList" model="@Model"/>
```

- Add a new view named `ByMake.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model IEnumerable<Car>
@{
  ViewData["Title"] = "Index";
}
<h1>Vehicle Inventory for @ViewBag.MakeName</h1>
@{
  var mode = new ViewDataDictionary(ViewData) {{"ByMake", true}};
}
<partial name="Partials/_CarList" model="Model" view-data="@mode"/>
```

Step 4: Create the Details view

- Add a new view named `Details.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model Car
@{
  ViewData["Title"] = "Details";
}
<h1>Details for @Model.PetName</h1>
@Html.DisplayForModel()
<hr/>
@*@Html.DisplayForModel("CarWithColors")*@
<div>
  <item-edit item-id="@Model.Id"></item-edit>
  <item-delete item-id="@Model.Id"></item-delete>
  <item-list></item-list>
</div>
```

Step 5: Create the Create view

- Add a new view named `Create.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model Car
@{
  ViewData["Title"] = "Create";
}
<h1>Create a New Car</h1>
<hr/>
<form asp-controller="Cars" asp-action="Create">
  <div class="row">
    <div class="col-md-4">
      <div asp-validation-summary="ModelOnly" class="text-danger"></div>
      @Html.EditorForModel()
    </div>
  </div>
  <div class="d-flex flex-row mt-3">
    <button type="submit" class="btn btn-success">
        Create <i class="fas fa-plus"></i>
    </button>&nbsp;&nbsp;|&nbsp;&nbsp;
    <item-list></item-list>
  </div>
</form>
@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

Step 6: Create the Edit view

- Add a new view named `Edit.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model Car
@{
  ViewData["Title"] = "Edit";
}
<h1>Edit @Model.PetName</h1>
<hr />
<form asp-area="" asp-controller="Cars" asp-action="Edit" asp-route-id="@Model.Id">
  <div class="row">
    <div class="col-md-4">@Html.EditorForModel()</div>
  </div>
  <div class="d-flex flex-row mt-3">
     <input type="hidden" asp-for="Id" />
     <input type="hidden" asp-for="TimeStamp" />
     <button type="submit" class="btn btn-primary">Save <i class="fas fa-
save"></i></button>&nbsp;&nbsp;|&nbsp;&nbsp;
     <item-list></item-list>
  </div>
</form>
@section Scripts {
  <partial name="_ValidationScriptsPartial" />
}
```

Step 7: Create the Delete view

- Add a new view named `Delete.cshtml` to the `Views\Cars` folder and update the markup to the following:

```
@model Car
@{
  ViewData["Title"] = "Delete";
}
<h1>Delete @Model.PetName</h1>
<h3>Are you sure you want to delete this car?</h3>
<div>
    @Html.DisplayForModel()
    <form asp-action="Delete">
      <input type="hidden" asp-for="Id" />
      <input type="hidden" asp-for="TimeStamp" />
      <button type="submit" class="btn btn-danger">Delete <i class="fas fa-
trash"></i></button>&nbsp;&nbsp;|&nbsp;&nbsp;
      <item-list></item-list>
  </form>
</div>
```

Summary

In this lab, you created the `BaseCrudController` and finished the Cars Controller. Then the lab created and or updated the views for the main application.

Next steps

The next lab creates custom validation attributes.