Walkthrough 14 - Fully Implementing a REST API

Setup

This lab will explore further explore Web API.

1. Start SQL Server.
2. Start Visual Studio.
3. Click Create a new project.
4. Set language to **C#** and project type to **Web**.
5. Select the ASP.NET Core Web API template, click Next.
6. Set Project name to **PhysicianAPI**.
7. Set Location to **a folder of your choosing**.
8. Set Solution name to **MedicalStaff**.
9. Ensure Place solution and project in the same directory is unchecked, click Next.
10. Set version to **.NET 5.0**, set Authentication Type to **None**, unselect Configure for HTTPS, select Enable OpenAPI support, if necessary, click Create.
11. Delete WeatherForecast.cs and Controllers / WeatherForecastController.cs.

appsettings.json

1. Add a connection string for the CHDB database.
2.
```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "CHDB": "Server=localhost\\sqlexpress;Database=CHDB;Trusted_Connection=True"
  }
}
```
3. Save the file.

Scaffold-DbContext

1. Open the Package Manager Console (PMC) and issue the following commands.
2. **Install-Package Microsoft.EntityFrameworkCore.Tools**
3. **Install-Package Microsoft.EntityFrameworkCore.SqlServer**
4. **Scaffold-DbContext name=CHDB Microsoft.EntityFrameworkCore.SqlServer -OutputDir Models -Tables physicians**

Startup.cs

1. Register the database context. Add the **using PhysicianAPI.Models;** and **using Microsoft.EntityFrameworkCore;** directives.
2.
```
public void ConfigureServices(IServiceCollection services)
{

    services.AddControllers();
    services.AddDbContext<CHDBContext>(options =>
        options.UseSqlServer(Configuration.GetConnectionString("CHDB")));
    services.AddSwaggerGen(c =>
    {
        c.SwaggerDoc("v1", new OpenApiInfo { Title = "PhysicianAPI", Version = "v1" });
    });
}
```
3. Save the file.

PhysiciansController.cs

1. Right-click Controllers folder and select Add / Controller... .
2. Select API Controller with actions, using Entity Framework, click Add.
3. Set Model class to **Physician (PhysicianAPI.Models)**.
4. Set Data context class to **CHDBContext (PhysicianAPI.Models)**.
5. Accept the default Controller name **PhysiciansController**, click Add.
6. Update PostPhysician to compute the new physician's id.
7.
```
public async Task<ActionResult<Physician>> PostPhysician(Physician physician)
{
    var maxId = _context.Physicians.Max(p => p.PhysicianId);
    physician.PhysicianId = maxId + 1;

    _context.Physicians.Add(physician);
    try
```

```
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateException)
    {
        if (PhysicianExists(physician.PhysicianId))
        {
            return Conflict();
        }
        else
        {
            throw;
        }
    }

    return CreatedAtAction("GetPhysician", new { id = physician.PhysicianId }, physician);
}
```

8. Run the site. Click the first GET button (the one that doesn't have an id parameter). Click the Try it out button. Click the Execute button to see all physicians.
9. Click the GET button to collapse it.
10. Click the second GET button (the one that does have an id paremeter). Click the Try it out button. Set the id parmeter to **1**, click Execute.
11. Make note of the Request URL your API is using, it will be needed later; specifically the port.
12. Select the JSON in the Response body and click Ctrl+C to copy it.

Doctors Project Setup

1. Right-click the Solution in the Solution Explorer and select Add / New Project... .
2. Set language to **C#** and project type to **Web**.
3. Select the ASP.NET Core Web App (Model-View-Controller) template, click Next.
4. Set Project name to **Doctors**, leave Location as is, click Next.
5. Set version to **.NET 5.0**, set Authentication Type to **None**, unselect Configure for HTTPS, click Create.

Doc.cs

1. Create a new model file named **Doc** in the Models folder.
2. Delete the Doc class.
3.
4.
```
namespace Doctors.Models
{
    public class Doc
    {
    }
}
```

5. Place the cursor between the curly braces. From the Edit menu, select Paste Special / Paste JSON As Classes. Rename the class from RootObject to **Doc**.
6.
```
public class RootObjectDoc
{
    public int physicianId { get; set; }
    public string firstName { get; set; }
    public string lastName { get; set; }
    public string specialty { get; set; }
    public string phone { get; set; }
    public int ohipRegistration { get; set; }
}
```

7. Annotate the class, adding the **using System.ComponentModel.DataAnnotations;** directive.
8.
```
public class Doc
{
    [Key]
    public int physicianId { get; set; }

    [Display(Name = "First Name")]
    public string firstName { get; set; }

    [Display(Name = "Last Name")]
    public string lastName { get; set; }

    [Display(Name = "Specialty")]
    public string specialty { get; set; }

    [Display(Name = "Phone")]
    public string phone { get; set; }

    [Display(Name = "OHIP Registration")]
    public int ohipRegistration { get; set; }
}
```

9. Save the file.

ErrorViewModel.cs

1. In the Models folder, open ErrorViewModel class. Add a description property to it.
2.
```
public class ErrorViewModel
{
    public string RequestId { get; set; }
```

```
    public bool ShowRequestId => !string.IsNullOrEmpty(RequestId);

    public string Description { get; set; }
}
```

3. Save the file.

Error.cshtml

1. Open Views / Shared / Error.csthml. Add the description property and remove the Development Mode text.
2.
```
@model ErrorViewModel
@{
    ViewData["Title"] = "Error";
}

<h1> class="text-danger">Error.</h1>
<h2> class="text-danger">An error occurred while processing your request.</h2>

@if (Model.ShowRequestId)
{
    <p><strong>Request ID:</strong> <code>@Model.RequestId</code></p>
}

<p><strong>Description:</strong> <code>@Model.Description</code></p>
<h3>Development Mode</h3>
<p>
    Swapping to <strong>Development</strong> environment will display more detailed information about the error that occurred.
</p>
<p>
    <strong>The Development environment shouldn't be enabled for deployed applications.</strong>
    It can result in displaying sensitive information from exceptions to end users.
    For local debugging, enable the <strong>Development</strong> environment by setting the <strong>ASPNETCORE_ENVIRONMENT</strong> environment variable to <strong>Development</strong>
    and restarting the app.
</p>
```

3. Save the file.

DocsController.cs

1. Scaffold a new MVC Controller with read/write actions, name it **DocsController**.
2. Setup a string with the URL for the API with a trailing slash using the port noted earlier. Also, setup a string for application/json as it will be needed multiple times.
3.
```
public class DocsController : Controller
{
    private string baseUrl = "http://localhost:12345/api/";
    private string appJson = "application/json";

    // GET: Docs
    public ActionResult Index()
    ...
```

Index

1. Change the method signature of Index to be asynchronous.
2.
```
public async Task<ActionResult> Index()
```

3. Add the **using Doctors.Models;**, **using System.Net.Http;**, **using System.Net.Http.Headers;** and **using System.Text.Json;** directives.
4. Instantiate a list of Doc and set up a call to the web server.
5. Call the web server and get the JSON.
6. Parse the JSON into the list of Doc and return them in the view.
7.
```
public async Task<ActionResult> Index()
{
    try
    {
        var docs = new List<Doc>();

        using (var client = new HttpClient())
        {
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue(appJson));
            client.BaseAddress = new Uri(baseUrl);
            var response = await client.GetAsync("physicians");

            if (response.IsSuccessStatusCode)
            {
                var json = await response.Content.ReadAsStringAsync();
                docs = JsonSerializer.Deserialize<List<Doc>>(json);
            }
        }

        return View(docs);
    }
    catch (Exception ex)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = HttpContext.TraceIdentifier,
                Description = ex.Message
            });
```

```
    }
}
```

8. Right-click in the Index method and select Add View... select Razor View, click Add.
9. Set View name to **Index**, Template to **List**, Model class to **Doc (Doctors.Models)**, click Add.

Index.cshtml

1. Update the title and heading. Remove the physician id from the table and update the Edit, Details and Delete links.
2.
```
@model IEnumerable<Doctors.Models.Doc>

@{
    ViewData["Title"] = "~~Index~~Doctors";
}

<h1>~~Index~~@ViewBag.Title</h1>

<p>
    <a asp-action="Create">Create New</a>
</p>
<table class="table">
    <thead>
        <tr>
            ~~<th>~~
                ~~@Html.DisplayNameFor(model => model.physicianId)~~
            ~~</th>~~
            <th>
                @Html.DisplayNameFor(model => model.firstName)
            ...
@foreach (var item in Model) {
        <tr>
            ~~<td>~~
                ~~@Html.DisplayFor(modelItem => item.physicianId)~~
            ~~</td>~~
            <td>
                @Html.DisplayFor(modelItem => item.firstName)
...
<td>
    @Html.ActionLink("Edit", "Edit", new { ~~/*~~ id = item.~~PrimaryKey */~~physicianId }) |
    @Html.ActionLink("Details", "Details", new { ~~/*~~ id = item.~~PrimaryKey */~~physicianId }) |
    @Html.ActionLink("Delete", "Delete", new { ~~/*~~ id = item.~~PrimaryKey */~~physicianId })
</td>
...
```

3. Save the file.

_Layout.cshtml

1. Open Views / Shared / _Layout.cshtml and modify the Home link to the Docs controller.
2.
```
...
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="~~Home~~Docs" asp-action="Index">~~Home~~Docs</a>
</li>
<li class="nav-item">
    <a class="nav-link text-dark" asp-area="" asp-controller="Home" asp-action="Privacy">Privacy</a>
</li>
...
```

3. Save the file.

Running Multiple Projects

1. In the Solution Explorer, right-click the Solution and select Properties.
2. Under Common Properties, select Startup Project.
3. Select **Multiple startup projects**.
4. Change the Action dropdown to **Start** for both projects.
5. Select the ProjectAPI project and click the Up button, so that it starts first.
6. Click OK.
7. Run the site, select the browser tab with the MVC page (not Swagger), if necessary. Click the Docs link. The physicians should appear.

DocsController.cs

GetDocAsync

1. Since the Details, Edit and Delete Gets will all be the same, create a new method named GetDocAsync to retrieve a doctor from the API.
2. Instantiate a Doc and set up a call to the web server.
3. Call the web server and get the JSON.
4. Parse the JSON into the Doc and return it.
5.
```
private async Task<Doc> GetDocAsync(int id)
{
    var doc = new Doc();

    try
    {
        using (var client = new HttpClient())
        {
```

```
            client.DefaultRequestHeaders.Accept.Clear();
            client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue(appJson));
            client.BaseAddress = new Uri(baseUrl);
            var response = await client.GetAsync($"physicians/{id}");

            if (response.IsSuccessStatusCode)
            {
                var json = await response.Content.ReadAsStringAsync();
                doc = JsonSerializer.Deserialize<Doc>(json);
            }
        }
    }
    catch
    {
    }

    return doc;
}
```

Details

1. Change the method signature of Details to be asynchronous, get the doctor and return it in the view.
2.
```
public async Task<ActionResult> Details(int id)
{
    var doc = await GetDocAsync(id);

    if (doc.physicianId == 0)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = id.ToString(),
                Description = $"Unable to find doctor with id={id}"
            });
    }

    return View(doc);
}
```

3. Right-click in the Details method and select Add View... select Razor View, click Add..
4. Set View name to **Details**, Template to **Details**, Model class to **Doc (Doctors.Models)**, click Add.

Details.cshtml

1. Remove the physician id and update the Edit link.
2.
```
...
<dl class="row">
    <dt> class = "col-sm-2">
        @Html.DisplayNameFor(model => model.physicianId)
    </dt>
    <dd> class = "col-sm-10">
        @Html.DisplayFor(model => model.physicianId)
    </dd>
    <dt> class = "col-sm-2">
        @Html.DisplayNameFor(model => model.firstName)
    </dt>
...
<div>
    @Html.ActionLink("Edit", "Edit", new { /* id = Model.PrimaryKey */physicianId }) |
    <a asp-action="Index">Back to List</a>
</div>
```

3. Save the file.
4. Run the site and access the details of a physician.

DocsController.cs

Create

1. Update the POST Create method signature to be asynchronous and accept a Doc object.
2.
```
public async Task<ActionResult> Create(IFormCollection collectionDoc doc)
```

3. If the model state is valid, serialize the Doc object into JSON, add the **using System.Text;** directive and set up a call to the web server.
4. Call the web server.
5.
```
public async Task<ActionResult> Create(Doc doc)
{
    try
    {
        if (ModelState.IsValid)
        {
            var content = new StringContent(JsonSerializer.Serialize(doc), Encoding.UTF8, appJson);

            using (var client = new HttpClient())
            {
                client.DefaultRequestHeaders.Accept.Clear();
                client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue(appJson));
                client.BaseAddress = new Uri(baseUrl);
                var response = await client.PostAsync("physicians", content);
            }
        }

        return RedirectToAction(nameof(Index));
```

```
    }
    catch (Exception ex)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = HttpContext.TraceIdentifier,
                Description = ex.Message
            });
    }
}
```

6. Right-click in the Create method and select Add View... select Razor View, click Add.
7. Set View name to **Create**, Template to **Create**, Model class to **Doc (Doctors.Models)**, click Add.

Create.cshtml

1. Remove the physician id and set autofocus to first name.
2.
```
...
<form asp-action="Create">
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    ~~<div class="form-group">~~
        ~~<label asp-for="physicianId" class="control-label"></label>~~
        ~~<input asp-for="physicianId" class="form-control" />~~
        ~~<span asp-validation-for="physicianId" class="text-danger"></span>~~
    ~~</div>~~
    <div class="form-group">
        <label asp-for="firstName" class="control-label"></label>
        <input asp-for="firstName" class="form-control" autofocus />
        <span asp-validation-for="firstName" class="text-danger"></span>
    </div>
    ...
```
3. Run the site and create a new physician.

DocsController.cs

Edit

1. Change the GET Edit method signature to be asynchronous, get the doctor and return it in the view.
2.
```
public async Task<ActionResult> Edit(int id)
{
    var doc = await GetDocAsync(id);

    if (doc.physicianId == 0)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = id.ToString(),
                Description = $"Unable to find doctor with id={id}"
            });
    }

    return View(doc);
}
```
3. Update the POST Edit method signature to be asynchronous and accept a Doc object.
4.
```
public async Task<ActionResult> Edit(int id, ~~IFormCollection collection~~Doc doc)
```
5. If the model state is valid, serialize the Doc object into JSON and set up a call to the web server.
6. Call the web server.
7.
```
public async Task<ActionResult> Edit(int id, Doc doc)
{
    try
    {
        if (ModelState.IsValid)
        {
            var content = new StringContent(JsonSerializer.Serialize(doc), Encoding.UTF8, appJson);

            using (var client = new HttpClient())
            {
                client.DefaultRequestHeaders.Accept.Clear();
                client.DefaultRequestHeaders.Accept.Add(new MediaTypeWithQualityHeaderValue(appJson));
                client.BaseAddress = new Uri(baseUrl);
                var response = await client.PutAsync($"physicians/{id}", content);

                if (!response.IsSuccessStatusCode)
                {
                    return View("Error",
                        new ErrorViewModel
                        {
                            RequestId = HttpContext.TraceIdentifier,
                            Description = response.StatusCode.ToString()
                        });
                }
            }
        }

        return RedirectToAction(nameof(Index));
    }
    catch (Exception ex)
    {
        return View("Error",
            new ErrorViewModel
            {
```

```
                RequestId = HttpContext.TraceIdentifier,
                Description = ex.Message
            });
    }
}
```

8. Right-click in the Create method and select Add View... select Razor View, click Add.
9. Set View name to **Edit**, Template to **Edit**, Model class to **Doc (Doctors.Models)**, click Add.

Edit.cshtml

1. Hide the physician id and set autofocus to first name.
2.
```
...
<form asp-action="Edit">
    <div> asp-validation-summary="ModelOnly" class="text-danger"></div>
    <div class="form-group">
        <label> asp-for="physicianId" class="control-label"></label>
    <input asp-for="physicianId" class="form-control" type="hidden" />
        <span> asp-validation-for="physicianId" class="text-danger"></span>
    </div>
    <div class="form-group">
        <label> asp-for="firstName" class="control-label"></label>
        <input asp-for="firstName" class="form-control" autofocus />
        <span> asp-validation-for="firstName" class="text-danger"></span>
    </div>
    ...
```

3. Run the site and edit the recently created new physician.

DocsController.cs

Delete

1. Change the method signature of GET Delete to be asynchronous, get the doctor and return it in the view.
2.
```
public async Task<ActionResult> Delete(int id)
{
    var doc = await GetDocAsync(id);

    if (doc.physicianId == 0)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = id.ToString(),
                Description = $"Unable to find doctor with id={id}"
            });
    }

    return View(doc);
}
```

3. Update the POST Delete method signature to be asynchronous.
4.
```
public async Task<ActionResult> Delete(int id, IFormCollection collection)
```

5. Set up a call to the web server.
6. Call the web server.
7.
```
public async Task<ActionResult> Delete(int id, IFormCollection collection)
{
    try
    {
        using (var client = new HttpClient())
        {
            client.BaseAddress = new Uri(baseUrl);
            var response = await client.DeleteAsync($"physicians/{id}");

            if (!response.IsSuccessStatusCode)
            {
                return View("Error",
                    new ErrorViewModel
                    {
                        RequestId = HttpContext.TraceIdentifier,
                        Description = response.StatusCode.ToString()
                    });
            }
        }

        return RedirectToAction(nameof(Index));
    }
    catch (Exception ex)
    {
        return View("Error",
            new ErrorViewModel
            {
                RequestId = HttpContext.TraceIdentifier,
                Description = ex.Message
            });
    }
}
```

8. Right-click in the Delete method and select Add View... select Razor View, click Add.
9. Set View name to **Delete**, Template to **Delete**, Model class to **Doc (Doctors.Models)**, click Add.

Delete.cshtml

1. Remove the physician id.
2.
```
...
<dl class="row">
    <dt class = "col-sm-2">
        @Html.DisplayNameFor(model => model.physicianId)
    </dt>
    <dd class = "col-sm-10">
        @Html.DisplayFor(model => model.physicianId)
    </dd>
    <dt class = "col-sm-2">
        @Html.DisplayNameFor(model => model.firstName)
    </dt>
    ...
```
3. Run the site and delete the recently created new physician.