Blazor

ASP520

Module 2

Implementing & Using Services

1 Implementing a Service

1.1 Data Model

Not everything has to be stored in a database. A web application may need to display information that usually does not change very often. This information can be stored in an external data file so that it can still be changed easily without touching any pages and components and having to rebuild the application. If we do not wish this file to be directly accessible from the web browser, do not put it into web root folder. Create a **Content** folder instead and add the following JSON file into the folder.

JSON file containing office information: Content/offices.json

```
[
    {
        "Id"        : "Penang (mainland) office",
        "Address1"  : "484B, 2nd Floor, Jalan Permatang Rawa,",
        "Address2"  : "Bandar Perda, Bukit Mertajam,",
        "Address3"  : "14000, Penang, Malaysia.",
        "Email"     : "info@newrise.com.my",
        "Tel"       : "604-6042307",
        "Mobile"    : "016-4422957"
    },
    {
        "Id"        : "Penang (island) office",
        "Address1"  : "2-3-11, Bangunan Lip Sin,",
        "Address2"  : "Lebuh Pekaka 1, Sungai Dua,",
        "Address3"  : "11700 Gelugor, Penang, Malaysia.",
        "Email"     : "info@newrise.com.my",
        "Tel"       : "604-6042308",
        "Mobile"    : "012-4186092"
    }
]
```

We will now create a data model class to represent a single entity, which in this case is an office. In a Blazor Server application this class is only required on the server but if you intend to run the application on the web browser later on, it will save you time to create models in a separate library so it can be used by both the server and client applications. Add in a new project named **Newrise.Shared** using Razor Class Library project template. Make sure to disable nullable setting in the project file. Then add a reference to the library from the **Newrise** project.

Project Information

```
Project name                    : Newrise.Shared
Project type                    : Razor Class Library
Location                        : <same_solution>
Target Framework                : .NET 8.0
Support pages and views         : uncheck
```

Disable nullable warnings:

```
<Project Sdk="Microsoft.NET.Sdk.Razor">
    <PropertyGroup>
        <TargetFramework>net8.0</TargetFramework>
        <Nullable>disable</Nullable>
        <ImplicitUsings>enable</ImplicitUsings>
    </PropertyGroup>
        :
</Project>
```

You can remove all existing files in the library project and then add a **Models** folder. Add the following model class into the folder that maps exactly to the structure of the objects in the JSON file. We will initialize the properties to empty string so that they will not be **null**.

A data model class: Newrise.Shared\Models\Office.cs

```
namespace Newrise.Shared.Models {
    public class Office {
        public string Id { get; set; } = string.Empty;
        public string Address1 { get; set; } = string.Empty;
        public string Address2 { get; set; } = string.Empty;
        public string Address3 { get; set; } = string.Empty;
        public string Email { get; set; } = string.Empty;
        public string Tel { get; set; } = string.Empty;
        public string Mobile { get; set; } = string.Empty;
    }
}
```

1.2 Service Implementing & Registration

We can also implement a service class to deserialize **Office** objects from the JSON file and expose the list of offices to the application. Since the data is only available on the server, this service will only be used by a Blazor server application so add a **Services** folder in the **Newrise** project instead and add a class named **OfficeListProvider** to it.

Class for loading a list of offices: Services\OfficeListProvider.cs

```
namespace Newrise.Services {
    public class OfficeListProvider {
    }
}
```

Some services depend on other services. Note that all dependent services needs to be registered first so the order of service registrations are important. You get get access to registered services by using constructor injection. Add one parameter for each type of service you need to use. Parameter type should be the class or interface used for service registration. Unless those services are only used in the constructor, you should declare fields to assign the services so that you can access them from any methods in your service. We need to use **IWebEnvironment** and **ILogger** services provided by ASP.NET Core so we will declare fields and assign the services to those fields from the constructor.

Getting access to other services

```
private readonly IWebHostEnvironment _environment;
private readonly ILogger<OfficeListProvider> _logger;

public OfficeListProvider(
    IWebHostEnvironment environment,
    ILogger<OfficeListProvider> logger) {
            _environment = environment;
            _logger = logger;
    }
}
```

The first thing we need to do is to construct the physical location of the JSON file. We can get the application root directory from environment **ContentRootPath** property while **WebRootPath** property will return the **wwwroot** directory instead.

Constructing physical path of JSON file

```
const string FileName = @"Content\offices.json";

private string _path;

public OfficeListProvider(
    IWebHostEnvironment environment,
    ILogger<OfficeListProvider> logger) {
    _environment = environment;
    _logger = logger;

    _path = Path.Combine(
        _environment.ContentRootPath, FileName);
}
```

We will add a **GetList** method to load and deserialize the JSON file to a **List<Office>** object. This is only done once as we can keep the object in a field to remember it and return back the same object the next time **GetList** is called.

Loading and deserializing the JSON file

```
private List<Office> _offices;

public List<Office> GetList() {
    if (_offices == null) {
        _offices = JsonSerializer.Deserialize<List<Office>>(File.ReadAllText(_path));
        _logger.LogInformation($"Office list loaded from '{_path}'.");
    }   return _offices;
}
```

We only need one instance of this object to load in the data. Once the data is loaded, it can be shared across multiple requests and users, so it makes sense to register the service as a singleton. There are three methods that you can use to register services with the container. The method used will determine how many objects are created, when it is created and the expected activation lifetime of the object.

Service registration methods

```
Scoped      AddScoped<>()       Object is created for each request
Singleton   AddSingleton<>()    Only one object is created per application
Transient   AddTransient<>()    Object created per injection call
```

A scoped object is used to process a single request so the object is no longer active at after the response is send back to the client. This ensures that no two requests can use the same object regardless of whether the requests are from one user or different users. Transient guarantees each component and page will not share the same object even if they are processing the same request from the same user. Singleton means a single object shared across requests and users.

Registering the service: Program.cs

```
services.AddSingleton<OfficeListProvider>();
```

1.3 Data Presentation

To make it easy to access the model and service, import the component namespaces in the **_Imports** component. Finally, create a **Contacts** component page to display a list of offices. Use property injection to get a **OfficeListProvider** singleton. Call the **GetList** method to access and return the list of offices. Override the **OnInitialized** method to get the office list after the component is loaded but before it is rendered.

Importing namespaces for models and services: _Imports.razor

```
        :
@using Newrise.Shared.Models;
@using Newrise.Services;
```

Retrieving office list in component page: ContactUs.razor

```
@page "/contacts"
@inject OfficeListProvider OfficeListProvider
<PageTitle>@(Program.AppName + "/Contact Us")</PageTitle>
<MudText Typo=Typo.h5>Contact Us</MudText>

@code {
    List<Office> offices;
    protected override void OnInitialized() {
        offices = OfficeListProvider.GetList();
    }
}
```

We will now use MudBlazor components to render data in the page. We use **MudGrid** and **MudItem** to arrange content across columns and rows. We used **MudPaper** to create a border container for each office. **MudDivider** can be used to insert a dividing line in the content. For blank lines you can just use the HTML **br** element. Since there could be multiple offices we use a **foreach** loop to repeat the content for each office. The result is not perfect yet but we just need to see the data at the moment.

Rendering office list in component page

```
<MudGrid>
@foreach(var office in offices) {
    <MudItem>
        <MudPaper>
            <MudText Typo=@Typo.h6 Align=Align.Center>@office.Id</MudText>
            <MudDivider />
            <MudText Typo=Typo.overline>ADDRESS</MudText>
            <MudText Typo=Typo.body1>@office.Address1</MudText>
            <MudText Typo=Typo.body1>@office.Address2</MudText>
            <MudText Typo=Typo.body1>@office.Address3</MudText>
            <br />
            <MudText Typo=Typo.overline>CONTACT</MudText>
            <MudText Typo=Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Email />
                <span>@office.Email</span>
            </MudText>
            <MudText Typo=Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Phone />
                <span>@office.Tel</span>
            </MudText>
            <MudText Typo=Typo.body1>
                <MudIcon Icon=@Icons.Material.Filled.Smartphone />
                <span>@office.Mobile</span>
            </MudText>
        </MudPaper>
    </MudItem>
}
</MudGrid>
```

1.4 Responsive Content

A responsive web application can change and re-arrange content to fit the screen size of a device. This make an application work well across desktop, notebook, tablet and smartphone devices. The first thing that you need to do is to add a viewport meta tag to the application hosting page to fix the width of the page to the width of the device. A responsive UI framework will then adapt the page content to the device width.

Adding viewport meta tag: App.razor

```
<html lang="en">
<head>
    <base href="~/" />
    <meta charset="utf-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        :
```

If you wish to disable scaling, also set **maximum-scale** to **1.0**. Most UI frameworks uses a 12-column based responsive grid. A **MudGrid** can have up to 12 columns but then each column may be too small to show their content effectively on small devices. On a small device we may only want one **MudItem** to appear on a row but on large devices we may want up to 4. The following are breakpoints that are available in MudBlazor that you can use to determine the content width of each item for different devices.

MudBlazor Breakpoints

```
xs - Extra Small < 600px
sm - Small < 960px
md - Medium < 1280px
lg - Large < 1920px
xl - Extra Large < 2560x
xx - Extra Extra Large >= 2560px
```

Setting up the responsive size of a MudItem

```
<MudItem xs="12" md="6" xl="3">
             :
</MudItem>
```

Use breakpoint properties to set the number of logical columns that the component will take up. For XS device and above, the component will take up all 12 columns which means that only a single office will appear on one row. For MD device and above, two offices can appear on one row, and on XL devices and above, you can have up to four offices on one row. Let us also turn the heading into a **MudItem** that occupies the entire row.

Move page heading into a MudItem in the grid

```
<MudItem xs="12">
    <MudText Typo=Typo.h5 Align=Align.Center>Contact Us</MudText>
</MudItem>
```

There are a few more other issues with the current presentation. The office details are too close to the border and each office is too close to one another. A UI Framework provides a set of utility style classes for customization and adjustment. This can be assigned to any HTML element using the **class** property. For MudBlazor component, use the **Class** property instead. We will use 2 utility class groups; **p** for spacing within the border (padding) and **m** for spacing before the border (margin). Use **a** for all, **l** for left, **r** for right, **t** for top, **b** for bottom, **x** for left and right, **y** for both top and bottom, followed by the size after the dash. You can also use **auto** for the size to be calculated based on available space.

Increasing top margin for MudText heading

```
<MudText Class="mt-4" Typo=Typo.h5 Align=Align.Center>Contact Us</MudText>
```

Increasing margin and padding for MudPaper

```
<MudPaper Class="ma-4 pa-4">
```

Alternatively you can change the CSS style attributes of each element by using **Style** property on a MudBlazor component or style attribute on a HTML element. Following shows changing the alignment of the text to better align with the icon.

Solving misaligned icon and text

```
<MudText Typo=@Typo.body1>
    <MudIcon Icon=@Icons.Material.Filled.Email />
    <span style="vertical-align:top">@office.Email</span>
</MudText>
```

You can always add custom style classes to simplify styling and changing styles. Add the class below to your existing application stylesheet and set **class** parameter for HTML and **Class** parameter for MudBlazor components.

Add a new style class: wwwroot\app.css

```
.icon-text {
    vertical-align:top;
}
```

Using style class

```
<MudText Typo=@Typo.body1>
    <MudIcon Icon=@Icons.Material.Filled.Email />
    <span class="icon-text">@office.Email</span>
</MudText>
```

Blazor supports isolated component styling. You can add a stylesheet for a component using the component name and placing it in the same location. Any style classes you add into the component stylesheet will only be used for that component and will not affect style classes for the application or other components even if they all have the same name.

Isolated component styling: ContactUs.razor.css

```
.icon-text {
    vertical-align:top;
}
```

Blazor will pre-process and combined all the component stylesheets into one with the project name followed by **.styles.css** that you include into your application as shown below.

Include component stylesheets: App.razor

```
<link rel="stylesheet" href="Newrise.styles.css" />
```

1.5 Object Caching

Some content are not accessed frequently, so you do not want to load them nor keep them in memory for a long period of time. Call the **AddMemoryCache** method to add caching service to your application.

Adding caching service: Program.cs

```
services.AddMemoryCache();
services.AddSingleton<OfficeListProvider>();
```

We will now update our **OfficeListProvider** to use caching. Declare **IMemoryCache** field and use constructor dependency injection to obtain the cache service. We do not need a field to assign the list as we will always fetch it from the cache.

Get access to memory cache service: OfficeListProvider.cs

```
private readonly IMemoryCache _cache;

public OfficeListProvider(
    IMemoryCache cache,
    IWebHostEnvironment environment,
    ILogger<OfficeListProvider> logger) {
    _cache = cache;
    _environment = environment;
    _logger = logger;
        :
}
```

Update the **GetList** method to first attempt to fetch the office list from the cache. If it does not exist only then we will load it in and cache it. We will use the filename as the cache key.

New GetList method that uses caching

```
// private List<Office> _offices;

public List<Office> GetList() {
    var offices = _cache.Get<List<Office>>(FileName);
    if (offices == null) {
        offices = JsonSerializer.Deserialize<List<Office>>(File.ReadAllText(_path));
        _logger.LogInformation($"Office list loaded from '{_path}'.");
        _cache.Set(FileName, offices);
    }   return offices;
}
```

IMemoryCache service provides many auto-expiration options. When a cached item is expired, it is not be returned and disposed instead. You can expire based on a fixed time period from the time you added the item to the cache. To set multiple expiration options including sliding expiration, use **MemoryCacheEntryOptions**.

Expire cached item in 15 minutes

```
_cache.Set(Filename, offices, TimeSpan.FromMinutes(15));
```

Expire cached item in 15 minutes from last access

```
_cache.Set(Filename, offices, new MemoryCacheEntryOptions {
    SlidingExpiration = TimeSpan.FromMinutes(15)});
```

The cached item will be considered as expired if the file has been modified. This will ensure that the file is loaded once on demand and does not need to loaded again until it is changed. You can of course manually remove the item from the cache by calling the **Remove** method. We will add a **Remove** method to our service if there is a need to manually remove the office list from the cache.

Manual remove of cached item

```
public void Remove() {
    _cache.Remove(Filename);
}
```

You can also remove the cached item when the file is changed, this ensures that users are always looking at the latest data. You can obtain an **IChangeToken** by calling the **Watch** method on the environment **ContentRootFileProvider** property.

Obtaining a change token

```
private IChangeToken _token;

IChangeToken GetToken() {
    return _token = _environment.ContentRootFileProvider.Watch(FileName);
}
```

You can then associate the change token with the item you placed into the cache. You no longer have to expire the cached item based on time since it will never be reloaded if the file has not been changed.

Registering cache item with token

```
_cache.Set(Filename, offices, GetToken());
```

If you still want to use time expiration as well, you can manually check the token and remove the cached item or simply include token check when retrieving a cached item as shown below. Make sure to get a new token after using it since **HasChanged** will remain true and will not reset.

Removing cached item

```
if (_token.HasChanged) { Remove(); GetToken(); }
```

Check token before retrieving cached item

```
var offices = null;
if (_token.HasChanged) GetToken();
else offices = _cache.Get<List<Office>>(FileName);
```

2 Implementing Database Services

2.1 Data Model

Using Entity Framework you can generate the data model from an existing database (database first) or generate a database from a manually designed data model (code first). For the project we will use a code first approach. First ensure that the following Entity Framework Core packages have been installed to the project.

Entity Framework Core packages

```
Microsoft.EntityFrameworkCore
Microsoft.EntityFrameworkCore.SqlServer
Microsoft.EntityFrameworkCore.Tools
```

We need to have a data model to represent the events organized by the company and users can sign up to participate in those events. First add an **EventType** enumeration to identify all the different types of events to be organized. Note we start from 1 so 0 can used for input validation as an indication that the user has not selected a value. Then add both an **Event** and the **Participant** classes as well.

Enumeration of event types: Models\EventType.cs

```
namespace Newrise.Shared.Models {
    public enum EventType {
        None = 0,
        Presentation,
        Training,
        Workshop,
        Forum
    }
}
```

Entity class for event: Models\Event.cs

```
namespace Newrise.Shared.Models {
    public class Event {
        public string Id { get; set; }
        public EventType Type { get; set; }
        public string Title { get; set; }
        public string Description { get; set; }
        public DateTime From { get; set; }
        public double Hours { get; set; }
        public DateTime To { get { return From.Add(TimeSpan.FromHours(Hours)); }}
        public int Seats { get; set; }
        public int AllocatedSeats { get; set; }
        public int RemainingSeats { get { return Seats - AllocatedSeats; }}
        public decimal Fee { get; set; }
        public bool Online { get; set; }
    }
}
```

Entity class for event participant: Models\Participant.cs

```
namespace Newrise.Shared.Models {
    public class Participant {
        public string Id { get; set; }
        public string Name { get; set; }
        public string Company { get; set; }
        public string Position { get; set; }
        public string Email { get; set; }
        public byte[] Photo { get; set; }
    }
}
```

In Entity Framework relationships can be formed between entities in a data model by using virtual properties. One-to-one, one-to-many or many-to-many relationships can be formed depending on whether type of property represents one entity or an entity collection. In our project one event can have multiple participants and one participant can participate in multiple events. Thus a many-to-many relationship will be required. Always use the **ICollection<T>** as property type but actual collection class can be **List<T>** or **HashSet<T>** if duplicates are not allowed. For example each participant cannot register more than once for each event. So every participant in one event will be unique and every event the participant is registered for is unique.

Each event can have multiple participants: Models\Event.cs

```
public class Event {
            :
    public virtual ICollection<Participant> Participants { get; set; } =
        new HashSet<Participant>();
}
```

Each participant can register for multiple events: Models\Participant.cs

```
public class Participant {
            :
    public virtual ICollection<Event> Events { get; set; } = new HashSet<Event>();
}
```

To create a new database or create tables in an existing database from the model you need to implement a **DbContext** class. Declare a **DbSet<T>** property for each entity type to generate a table in the database. Add a constructor to accept a configuration object and pass it to the base class. This object will provide configuration information such as the data provider to use and the connection string.

Database service class: Services\NewriseDbContext.cs

```
namespace Newrise.Services {
    public class NewriseDbContext : DbContext {
        public DbSet<Event> Events { get; set; }
        public DbSet<Participant> Participants { get; set; }
        public NewriseDbContext(DbContextOptions<NewriseDbContext> options) : base(options) {}
    }
}
```

2.2 Data Annotations

Our current data model does not provide enough details for the Entity Framework to create an optimal database. It will try to detect the primary key and the datatype for each column. However, all string columns will be nullable except for the primary key and the maximum length cannot be set as this information is not available from the data model. You can attach data annotations to the data model to provide additional information, not only for improving or customizing database generation but can also used for model validation and presentation.

Some data annotation attributes

Key	*Primary key - generation only*
DataType	*Customize the data type – generation only*
Required	*Cannot be null / blank – generation & validation*
StringLength	*String maximum and minimum length – generation & validation*
EmailAddress	*Matches standard email pattern – validation only*
RegularExpression	*Matches provided pattern – validation only*
Range	*Minimum to maximum value – validation only*

The following are data annotation attributes that we will assign to **Event** members. If you want to use the data model as a view model for input and presentation purposes then **ErrorMessage** parameter is important for showing validation error messages.

Annotations for Event entity class: Models/Event.cs

```
[Key, Required(ErrorMessage = "{0} is required.")]
[StringLength(6, ErrorMessage ="{0} can only have {1} characters.")]
[RegularExpression(@"^[A-Z]{3}\d{3}$", ErrorMessage ="{0} is not correctly formatted.")]
public string Id { get; set; }

[Range(1, 4, ErrorMessage = "{0} is not valid.")]
public EventType Type { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(50, ErrorMessage = "{0} can only have {1} characters.")]
public string Title { get; set; }

[StringLength(1000, ErrorMessage = "{0} can only contain {1} characters.")]
public string Description { get; set; }

[Range(0.5, 8.0, ErrorMessage = "{0} must be from {1} to {2}.")]
public double Hours { get; set; }

[Range(1, 200, ErrorMessage = "{0} must be from {1} to {2}.")]
public int Seats { get; set; }

[Range(0, 5000, ErrorMessage = "{0} must be from {1} to {2}.")]
public decimal Fee { get; set; }
```

The following are the data annotation attributes added for the **Participant** members. You can also write code to customize database generation by overriding the inherited **OnModelCreating** method and use the builder to customize columns as well as add constraints and indexes. Use **HasAlternateKey** method to create a unique constraint on the **Email** column of **Participant** and customize precision of **Fee** column.

Annotations for Participant entity: Models\Participant.cs

```
[Key, Required(ErrorMessage = "{0} is required.")]
[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Id { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Name { get; set; }

[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Company { get; set; }

[StringLength(40, ErrorMessage = "{0} can only have {1} characters.")]
public string Position { get; set; }

[Required(ErrorMessage = "{0} is required.")]
[StringLength(254, ErrorMessage = "{0} can only have {1} characters.")]
[EmailAddress(ErrorMessage = "{0} is not correctly formatted.")]
public string Email { get; set; }
```

Customize database generation: Models\NewriseDbContext.cs

```
protected override void OnModelCreating(ModelBuilder builder) {

    builder.Entity<Participant>().HasAlternateKey(p => p.Email);
    builder.Entity<Event>().Property("Fee").HasColumnType("decimal").HasPrecision(7, 2);
}
```

2.3 Database Migrations

It is time to apply our model to a new database or existing database. First construct the connection string to reference the database. Then call a **AddDbContextFactory** to register a factory to create **NewriseDbContext** instances. Setup the configuration options to use SQL Server and pass in the connection string to use.

Registering and configuring a DbContext factory: Program.cs

```
var connectionString =
    "Server=.;Database=NewriseDb;TrustServerCertificate=True;Trusted_Connection=True";
builder.Services.AddDbContextFactory<NewriseDbContext>(
    options => options.UseSqlServer(connectionString));
```

Open up the **Package Manager Console** and then type in the following command to create and name your initial database migration. This will be created in a **Migrations** folder. It generates a class so you can actually go through the generated code to see if it is correct. The **Up** method shows what operations will take place when migration is applied to the database and the **Down** method shows the operations to revert the migration. When you are ready to apply the migration to the database, you can use a **Update-Database** command. It will run the **Up** method in the migration class.

Add a database migration

```
Add-Migration InitialCreate
```

Apply the latest migration

```
Update-Database
```

If you have not applied the migration, you can simply delete the migration class. Use **Remove-Migration** command to revert an already applied migration. This will then run the **Down** method in the migration class.

Reverting an applied migration

```
Remove-Migration
```

If a migration is used to create a new database, you can just delete the migration and use the following command to delete the database. You would not need to revert any migration for a new data model and database.

Deleting new database

```
Drop-Database
```

The connection string that we are using is only for local development and probably is not the same for testing or production environment. The location of the database will not be the same for development, testing and production. Secondly, you wouldn't be using a trusted connection for production since you wouldn't be creating a **Windows** account for every user in the world. Also connections cannot be recycled across users that are using their individual Windows account to connect to the database even when connection pooling is enabled. Data access performance is greatly reduced when new connections have to be opened and authenticated for every request.

The best method is to store the connection string externally in the **appsettings.json** configuration file. Note that each runtime environment has its own configuration file to be merged with the main configuration file. So for a development only connection strings, store them in **appsettings.Development.json** file instead. Add them to the **ConnectionStrings** section and give each one a key. It is common to use a database name as the connection string key.

Development only configuration: appsettings.Development.json

```
{
   "ConnectionStrings": {
        "NewriseDb": "Server=.;Database=NewriseDb;TrustServerCertificate=True;Trusted_Connection=True"
    },
         :
}
```

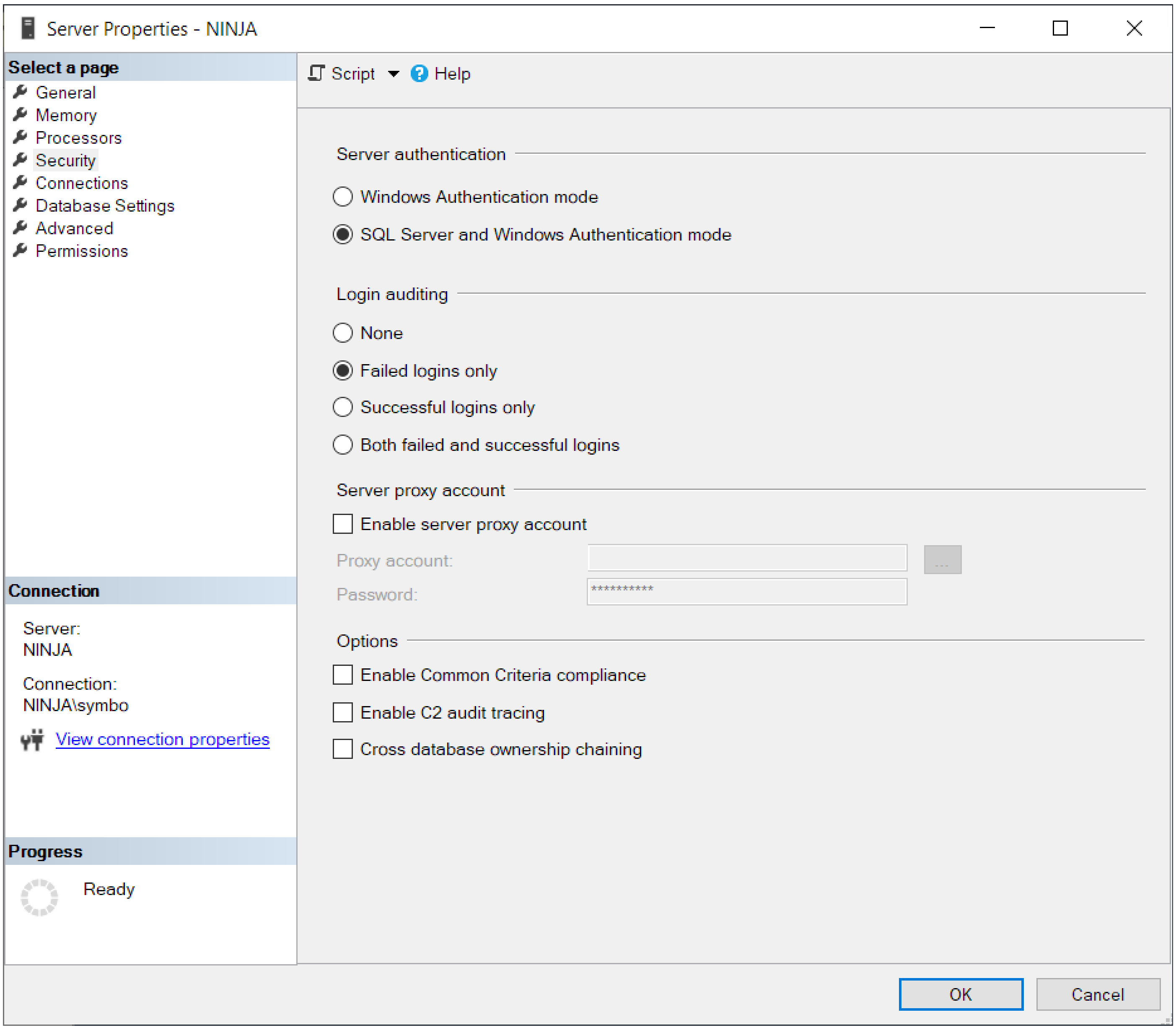
You can now access the connection string by calling a **GetConnectionString** method from **Configuration** property of the application host builder based on the what is the current environment.

Retrieving connection string from configuration file: Program.cs

```
var connectionString = builder.Configuration.GetConnectionString("NewriseDb");
builder.Services.AddDbContextFactory<NewriseDbContext>(options =>
    options.UseSqlServer(connectionString));
```

Testing is not only done by the development team members but can also be company staff or external end users. It would be better for to use SQL Server authentication instead of Windows authentication to access the database in testing and production environments. First you need to make sure SQL Server authentication is enabled as it is disabled by default when you initially install the SQL Server database engine using default settings. You can do this in SQL Server Management Studio. Right-click on the server in **Object Explorer** and select **Properties** option. Switch to **Security** page to see the authentication options.

Retrieving connection string from configuration file: Program.cs


Server Properties - NINJA

Select a page: General, Memory, Processors, Security, Connections, Database Settings, Advanced, Permissions

Server authentication
- Windows Authentication mode
- [✓ SQL Server and Windows Authentication mode]

Login auditing
- None
- [✓ Failed logins only]
- Successful logins only
- Both failed and successful logins

Server proxy account
- Enable server proxy account
- Proxy account:
- Password: **********

Options
- Enable Common Criteria compliance
- Enable C2 audit tracing
- Cross database ownership chaining

Connection
Server: NINJA
Connection: NINJA\symbo

Progress: Ready

You can then go to the **Logins** under **Security** node in **Object Explorer** to create or manage existing SQL Server accounts. There is an administrative account named **sa** but it has been disabled. Enabled it and set the password if you want to use it instead of using your Windows account to manage SQL Server. However do not use it for your application. You should always create a separate SQL Server accounts for applications and set just enough permissions for the application to accomplish its data tasks.

For testing and production databases, add a new SQL Server login using the following information and map that login into the application database and assign it the security roles for that database. Assign the connection string to the main configuration file so it becomes the default for non-development environments. Use User Id and Password fields to specify the account name and password in the connection string.

New SQL Server account

```
Name            :   NewriseUser
Password        :   AF21631253214D62920AAD8C4F453BC0
User Mapping    :   NewriseDb
                    Database Roles:
                        db_datareader
                        db_datawriter
```

Setting the default connection string for non-development: appsettings.json

```
{
    "ConnectionStrings": {
        "NewriseDb" : "Server=NINJA;Database=NewriseDb;TrustServerCertificate=True;
User Id=NewriseUser;Password=AF21631253214D62920AAD8C4F453BC0"
    },
            :
```

2.4 Data Service

We want to isolate as much processing code as possible away from the UI layer. The UI layer should concentrate only on presentation and input. Input should include the validation of input data. There's also a need to be able to abstract code that has to run on a server in case the UI runs on the client rather than on the server. This would make it difficult to convert a Blazor Server project to a Blazor Webassembly project if the UI layer is directly accessing the database.

We will now implement a separate data service class to access the database through the entity framework model we have created. In the future we can easily abstract the functionality of this class by using an interface. The class will use constructor injection to get access to the **DbContext** factory for Newrise.

Access to DbContext factory: Services\EventDataService.cs

```
namespace Newrise.Services {
    public class EventDataService {
        readonly IDbContextFactory<NewriseDbContext> _dcFactory;
        public EventDataService(IDbContextFactory<NewriseDbContext> dcFactory) {
            _dcFactory = dcFactory;
        }
    }
}
```

When we need to access the database, we will use the factory to create an instance of **NewriseDbContext**. We need to make sure the instance is disposed to make sure that the database connection returns back to the connection pool after use. Objects that has a **Close** or **Dispose** method need these methods to be called after use so the resources used my the method are released back to the system immediately.

Even though the resources will definitely be released when the object is collected by the Garbage Collector, you have no idea when it will run. If the object is not collected, it will hold on to resources that cannot be recycled for another process or user. You must make sure the object is disposed regardless of success or failure. We can do this through a **try finally** block.

Ensuring DbContext is disposed after use

```
public void AddEvent(Event item) {
    var dc = _dcFactory.CreateDbContext();
    try {
        dc.Events.Add(item);
        dc.SaveChanges();
    }
    finally {
        dc.Dispose();
    }
}
```

There is a simpler way of writing the above code, you can use a **using** statement to create an object and use that object within the **using** code block. When the block is exited, regardless of success or failure, the object will be disposed. The following code shows the correct way of creating and using a **DbContext** object to add an **Event** to the database. We can also implement an asynchronous version of the method.

Using C# auto-dispose feature

```
public void AddEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        dc.Events.Add(item);
        dc.SaveChanges();
    }
}
```

Asynchronous version of the above method

```
public async Task AddEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        await dc.Events.AddAsync(item);
        await dc.SaveChangesAsync();
    }
}
```

Following are the synchronous and asynchronous methods to remove an **Event** from the database. We should always try to avoid runtime errors whenever possible. There are many ways to ensure users are least likely to make mistakes in the UI layer.

Removing an existing Event

```
public void RemoveEvent(string id) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var item = dc.Events.Find(id);
        if (item != null) {
            dc.Events.Remove(item);
            dc.SaveChanges(); }
    }
}
```

Asynchronous version

```
public async Task RemoveEventAsync(string id) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var item = await dc.Events.FindAsync(id);
        if (item != null) {
            dc.Events.Remove(item);
            await dc.SaveChangesAsync();
        }
    }
}
```

You cannot directly use **SaveChanges** on one **DbContext** to update a changed entity retrieved using another DbContext. There are two ways to resolve this; re-fetch the entity and transfer the properties from a previous entity or re-attach a previous entity to the current DbContext and set its **EntityState** and **SaveChanges** will then try to update the database according to that state.

Refetch entity and update its properties

```
public void UpdateEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var source = dc.Events.Find(item.Id);
        if (source != null) {
            source.Type = item.Type;
            source.Title = item.Title;
            source.Description = item.Description;
            source.From = item.From;
            source.Hours = item.Hours;
            source.Seats = item.Seats;
            source.Fee = item.Fee;
            source.Online = item.Online;
            dc.SaveChanges();
        }
    }
}
```

Reattach the previous entity and update its EntityState

```
public void UpdateEvent(Event item) {
    using (var dc = _dcFactory.CreateDbContext()) {
        var entry = dc.Events.Attach(item);
        entry.State = EntityState.Modified;
        dc.SaveChanges();
    }
}
```

Asynchronous version of the above method

```
public async Task UpdateEventAsync(Event item) {
    using (var dc = await _dcFactory.CreateDbContextAsync()) {
        var entry = dc.Events.Attach(item);
        entry.State = EntityState.Modified;
        await dc.SaveChangesAsync();
    }
}
```

Returning all events as a List collection

```
public List<Event> GetEvents() {
    using (var dc = _dcFactory.CreateDbContext())
        return dc.Events.ToList();
}
```

Asynchronous version

```
public async Task<List<Event>> GetEventsAsync() {
    using (var dc = await _dcFactory.CreateDbContextAsync())
        return await dc.Events.ToListAsync();
}
```

We will now register the data service class as a singleton. This object can be fetched by other services through constructor injection or by **Razor** components and pages through property injection.

Registering our data service

```
builder.Services.AddSingleton<EventDataService>();
```