

The .NET Stack Handbook

C#, VB, Databases, Web and the Operations Around Them

Peter Gill

Version 1.0-draft

Contents

Preface	1
What this book is	1
Who it is for	1
What is checked, and what is not	1
Conventions	1
Licence	2
 I The Language	 3
1 The Language: C# and VB	5
1.1 Variables	5
1.1.1 Integers	5
1.1.2 Strings	6
1.1.2.1 StringBuilder	6
1.1.3 Decimals	7
1.1.4 Booleans	7
1.1.5 Chars	8
1.1.6 DateTime	8
1.1.7 Doubles	9
1.1.8 Objects	9
1.2 Objects	10
1.2.1 Methods	12
1.3 Interfaces	13
 2 Asynchronous Work and Threads	 17
2.1 Async/Await	17
2.2 Threads	19
2.2.1 Locks	20
2.3 In Memory Work Queue	22
2.3.1 Task Queue	22
2.3.2 Thread Queue	23
 3 Structuring an Application	 25
3.1 IOC	25
3.1.1 Registering and resolving services	25
3.1.2 Lifetimes	26
3.1.3 Registering by convention	26
3.1.4 Why this makes testing easy	26
3.1.5 Other containers	27
3.2 Repository Pattern	27
3.2.1 Use a base abstract class that is passed a connection	27

3.2.2	No base abstract. Let individual repository classes do as they please	29
3.2.3	Do something with the repository classes	29
3.3	Events	30
3.3.1	Use built in EventHandler	30
3.3.2	Use custom delegate as event handler	31
3.3.3	Subscribe to the event	31
3.3.4	VB example of basic custom events	31
3.3.5	Create a custom event	32
3.3.6	Use the custom event	32
3.4	Nuget	33
3.4.1	Create a NuGet Package	33
3.4.2	Add NuGet source	33
3.4.3	Check if NuGet Source already Exists	33
3.4.4	Start fresh with just nuget.org	33
3.5	Testing and Coverage	33
3.5.1	NUnit and Coverlet	33
3.5.2	Other test frameworks	35

II User Interfaces 37

4	Desktop User Interfaces	39
4.1	Winforms	39
4.1.1	A form with a control and an event handler	39
4.1.2	Keep the UI thread responsive	40
4.1.3	Cross platform alternatives	41
4.2	Microsoft Maui	41
4.2.1	Pages and XAML	41
4.2.2	Dependency injection	42
4.2.3	Keeping the UI responsive	43
4.2.4	Alternatives	43
5	Crystal Reports	45
5.1	Set data using DataTables	45
5.2	Set report parameters	46
5.3	Move report objects	47
5.4	Export to pdf or other file type	47
5.5	CrystalCmd Server and Client	47

III Data 49

6	SQLite	51
6.1	SQLite	51
6.2	C# Examples	51
6.3	SQLite with Dapper	52
6.4	SQLite with Entity Framework Core	53
7	PostgreSQL	55
7.1	PostgreSQL - Install	55
7.2	PostgreSQL Examples	55
7.2.1	Create a Table	55
7.2.2	Insert Data	55
7.2.3	Stored Procedure	55
7.2.4	Stored Function	56

7.2.5	View	56
7.2.6	C# Example: Querying PostgreSQL	56
7.3	PostgreSQL with Dapper	57
7.4	PostgreSQL with Entity Framework Core	57
8	Microsoft SQL Server	59
8.1	Adventure Works	59
8.2	Create a database	60
8.3	Create a table	60
8.4	Alter a table	60
8.5	Create indexes	60
8.6	SELECT	61
8.7	INSERT	61
8.8	UPDATE	61
8.9	DELETE	61
8.10	Foreign Keys	61
8.11	JOIN	62
8.12	CTE	62
8.13	Stored Procedures	63
8.14	Stored Functions	63
8.15	Views	63
8.16	SQL - Install	64
8.16.1	SQL server windows install	64
8.16.2	SQL server linux install	64
8.16.3	SQL server extra configuration after the install	65
8.17	Reference - Admin	65
8.18	Client tools	66
8.19	SQL Profiler and Extended Events	66
8.20	SQL Query Store	67
8.21	SQL Watch	67
9	Redis	69
9.1	Ubuntu Install	69
9.1.1	Add redis password	69
9.1.2	Connect to password protected redis server	70
9.1.3	expose to the network	70
9.1.4	Restart on failure	70
9.1.5	TLS Connection Support	71
9.1.5.1	1. Generate SSL/TLS Certificates	71
9.1.5.2	2. Configure Redis to Use TLS	71
9.1.5.3	3. Restart Redis Server	72
9.2	Session	72
9.3	Cache	72
9.4	Publish and Subscribe	73
9.4.1	Using Redis Pub/Sub in C	73
9.5	Example: Redis Pub/Sub with Web Frontend and Worker Services	73
9.5.1	1. Web Frontend (ASP.NET Core + JavaScript)	74
9.5.2	2. Worker Service (C# Console App)	75
9.5.3	3. Completion Notification Service	75
9.6	Work and Message Queue with Redis in C	76
9.6.1	Producer Example (Enqueue Work)	76
9.6.2	Consumer Example (Worker)	76
10	Data Access in .NET	79

10.1	DbConnection	79
10.2	DbCommand	80
10.3	DataAdapters	81
10.4	DbTransaction	83
10.5	ORM - Dapper	85
10.6	ORM - Entity Framework	85
10.6.1	Entity Framework: Raw SQL Queries with FromSql and FromSqlInterpolated . . .	86
10.6.2	Entity Framework Core: Disabling Change Tracking	86
10.7	Database Migrations - Entity Framework Core	87
10.7.1	1. Add EF Core Tools	87
10.7.2	2. Create a Migration	87
10.7.3	3. Apply the Migration	87
10.7.4	4. Example Migration Class	87
10.7.5	5. Further Changes	88
10.8	Database Migration - FluentMigrator	88
10.8.1	Example: Creating a Table with Fluent Syntax	88
10.8.2	Example: Executing Raw SQL in a Migration	88
10.8.3	Running Migrations	89
10.9	Transactions and Isolation Levels	89
10.9.1	Common Isolation Levels	89
10.9.2	Isolation Level Descriptions	89
10.9.3	Example: Setting Isolation Level in SQL	90
10.9.4	Example: Setting Isolation Level in C	90
10.9.5	Summary Table	91
10.10	SQL Database Backup	91
IV	Web and Hosting	93
11	ASP.NET Core	95
11.1	Dependency Injection	95
11.2	MVC	96
11.3	Minimal API	98
11.4	Blazor	99
12	Containers, nginx and Kubernetes	101
12.1	Containers - Docker	101
12.1.1	Example: Dockerfile for a .NET Web Application	101
12.1.2	Build and Publish with Docker Buildx and SBOM	102
12.2	nginx	102
12.2.1	Tell ASP.NET Core it is behind a proxy	103
12.2.2	TLS with Let's Encrypt	103
12.2.3	Run the application as a service	103
12.3	Kubernetes	104
13	JavaScript in the Browser	105
13.1	Use htmx to Avoid Complicated JavaScript	105
13.2	fetch	105
13.2.1	call fetch - form-urlencoded	105
13.2.2	call fetch - application/json	106
13.2.3	Helper functions	106
13.3	async/await	107
13.4	jQuery	108
13.5	Kendo UI	108

V	Operations	109
14	Monitoring and Observability	111
14.1	Prometheus Ecosystem	111
14.1.1	Key Features	111
14.1.2	Service Discovery	111
14.1.3	Example: Exposing Metrics in .NET	112
14.1.4	Example: Blackbox Exporter Configuration	112
14.2	Grafana for Visualization	112
14.2.1	Example: Prometheus Data Source in Grafana	112
15	Build Pipelines	113
15.1	GitHub Actions	113
15.1.1	GH Action to Create Linux Packages	115
15.2	Jenkins	116
15.2.1	Jenkins Plugin Setup	116
15.2.2	Jenkins Dotnet Pipeline	116
A	Git Clients	119
A.1	Git Visual Studio	119
A.2	Git Rider	120
A.3	Tortoise Git	120
A.4	Github Desktop	120

Preface

What this book is

A working reference for building .NET software and running it once it is built, written by someone who does that for a living rather than someone documenting features in order.

It began as a single post on majorsilence.com. That post grew past five thousand lines, which is the point at which a post stops being a post. It is now fifteen chapters across five parts: the languages, the user interfaces, four databases and the .NET side of talking to them, the web stack, and the operations that keep the result alive.

Who it is for

Developers who already program, and who need the .NET answer to something specific. It does not teach programming from nothing, and it does not assume you have used .NET before.

The chapters are deliberately independent. Read one, take what you need, and leave. Where one chapter genuinely depends on another - and it happens most often with [Structuring an Application](#), whose repository and dependency injection patterns reappear everywhere afterwards - there is a link rather than a repeated explanation.

What is checked, and what is not

Every listing targets .NET 10 (`net10.0`) unless a section says otherwise.

The C# and VB code is not typed into the page and hoped for. It lives in an `examples/` solution alongside the text, and every build compiles it, runs the tests, and executes the console examples. When a chapter states what a program prints, that is the output it actually produced. Writing that solution found seven bugs in the prose it was copied from, which is a fair estimate of how many are in any programming book that lacks one.

The rest is not machine checked and should be read with correspondingly more suspicion. SQL run against a live server, Dockerfiles, nginx configuration, Kubernetes manifests and CI pipeline definitions are all reviewed but not executed by the build. Where that distinction matters, the text says so.

Tooling ages faster than anything else in a book like this. Recommendations here are current as of writing and will not stay that way; the underlying commands and configuration outlast the graphical tools that wrap them, which is why the text prefers them.

Conventions

Examples are Linux first, because that is where most .NET now runs in production. Where Windows differs in a way that matters, both are given.

C# and VB appear side by side in the early chapters. Later chapters are mostly C#, not out of any judgement about VB, but because repeating every listing twice would double the length of the book without teaching anything the first two chapters have not already shown.

Licence

The prose is licensed [CC BY-SA 4.0](#). The code samples, and everything in the `examples/` and `tools/` directories, are MIT.

That split is deliberate. Copy a listing into your own application freely, with no obligation beyond the copyright notice - a book that exists to be copied from should not force you to relicense your work for taking ten lines from it.

Part I

The Language

Chapter 1

The Language: C# and VB

C# and VB are two languages over one runtime. They compile to the same IL, use the same base class library, and can call each other freely inside one solution. Choosing between them is a matter of taste and of what the team already knows, not of capability.

This chapter covers the three things everything else is built from: variables to hold values, objects to group values with the code that operates on them, and interfaces to describe what an object can do without saying how. Both languages are shown side by side throughout, so a VB developer reading C# code, or the reverse, can see the same idea twice.

1.1 Variables

Variables are the basic working blocks in code. You use variables to hold values. There are several different variable types but in this lesson we will cover only four of them.

To declare a variable you use the language keyword “Dim” used with a name and “As”. So if you want a string called “Hello World” named TestVariable you would declare it like this.

```
Dim TestVariable As String = "Hello World"
```

```
string TestVariable = "Hello World";
```

This example declares a variable and assigns a value at the same time. However you can declare a variable without assigning value. The value can always be assigned later. A good general rule is only declare a variable when it is ready to be used (assigned) when possible.

- Integer - are like whole numbers but can contain negatives
- String - contain multiple characters
- Decimal - numbers with decimals
- Boolean - is true or false

1.1.1 Integers

Integers are like whole numbers but can contain negatives. So they have negatives, zero, or positives.

For example -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5 are all integer values.

To declare an integer you can do this.

```
Dim i As Integer
```

```
int i;
```

This example creates a new variable of type Integer name i. It does not assign any value to i. To assign a value to i you can do it like this.

```
i=1
```

```
i=1;
```

Now the value of i is 1.

```
i=2
```

```
i=2;
```

Now the value of i is 2.

1.1.2 Strings

Strings can hold any value. They can have letters, numbers, special characters. They can be long or short To declare a string you can do this.

```
Dim s As String
```

```
string s;
```

This example creates an empty string called s. This string has no value

To assign the value “hello world” to the variable s we would do.

```
s = "hello world"
```

```
s = "hello world";
```

The value of the variable can be reassigned at any time so if we want to change the value to “purple monkey dishwasher” just do the same as above put with the new string.

```
s = "purple monkey dishwasher"
```

```
s = "purple monkey dishwasher";
```

If we want to see the value of a variable printed to the console we can write.

```
System.Console.WriteLine("The value of s is: " & s)
```

```
System.Console.WriteLine($"The value of s is: {s}");
```

We see here that the value of s is appended to the string “The value of s is:” and then printed to the console as “The value of s is: purple monkey dishwasher”. You can append any string to any other string at any time using the & symbol in vb or the + symbol in c#.

1.1.2.1 StringBuilder

If you are appending to a string again and again or changing its value over and over again this can become very slow. String operations like this can be sped up using the StringBuilder class.

To use a string builder you need to initialize System.Text.StringBuilder.

```
Dim builder As New System.Text.StringBuilder
builder.Append("Hello World ")
builder.Append("Peter. ")
builder.Append("Have a good day.")
```

```
System.Console.WriteLine(builder.ToString)
```

```
var builder = new System.Text.StringBuilder();
builder.Append("Hello World ");
builder.Append("Peter. ");
builder.Append("Have a good day.");

System.Console.WriteLine(builder.ToString());
```

This will print “Hello World Peter. Have a good day.”.

What this does is keep adding to a buffer and when you call the ToString method it finally creates a string. This is much faster than concatenating the string together like the following.

```
System.Console.WriteLine("Hello World " & "Peter. " & "Have a good day.")
```

```
System.Console.WriteLine("Hello World " + "Peter. " + "Have a good day.");
```

This example probably is not faster since it is so tiny but if you did this with 100 000 strings the string builder would be much faster.

1.1.3 Decimals

Decimals are used when you need numeric values that contain decimal places. It is essential if you are doing financial calculations that you use decimals and no other data type. Do not use doubles.

For example -5.32, -4.76, -3.7654, -2.1, -1.343, 0.13, 1.786555, 2.2, 3.765, 4.22, 5.3446 are all decimal values.

To declare a decimal you can do this.

```
Dim d As Decimal = 4.444D
```

```
decimal d = 4.444m;
```

This example creates a new variable of type Decimal with the name d and a value of 4.444d

To assign a new value to d you can do it like this.

```
d=5.437D
```

```
d = 5.437m;
```

Now the value of d is 5.437.

```
d=2.55
```

```
d = 2.55m;
```

Now the value of d is 2.55. As shown above variables in function can always be reassigned new values

1.1.4 Booleans

Booleans are variables that can be either True or False. That is all they hold. Booleans default to false.

To declare a boolean you can do this.

```
Dim b As Boolean = False
```

```
bool b = false;
```

This example creates a new variable of type Boolean named b. It does not assign any value to b. To assign a value to b you can do it like this.

```
b=True
```

```
b=true;
```

Now the value of b is True.

```
b=False
```

```
b=false;
```

Now the value of b is False.

There is no other value that a boolean can hold. If you do not set a value a boolean will default to False.

1.1.5 Chars

Chars are variables that can hold one character and only one character. It can be any character available but only one character at a time.

To declare a char you do this.

```
Dim c As Char
```

```
char c;
```

This example creates a new variable of type Char named c. It does not assign any value to c. To assign a value to c you can do it like this.

```
c="A" c
```

```
c='A';
```

Now the value of c is A.

```
c="~" c
```

```
c='~';
```

Now the value of c is ~.

As is written above any character can be held in a char variable but only one character at a time. Like any other variable type you can print the variable to the console using like this.

```
System.Console.WriteLine(c)
```

```
System.Console.WriteLine(c);
```

1.1.6 DateTime

DateTime variables can hold a date and time value. If you just want a date you can also use Date instead of DateTime.

To declare a DateTime you do this.

```
Dim t As DateTime
```

```
DateTime t;
```

This example creates a new variable of type DateTime named t. It does not assign any value to t. To assign a value to t you can do it like this.

```
t = DateTime.Now
```

```
t = DateTime.Now;
```

This assigns the current date and time to the variable `t`.

If you want to assign a specific date such as 01 may 2012, do it like this.

```
t = New DateTime(2012, 5, 1)
```

```
t = new DateTime(2012, 5, 1);
```

Now the date of `t` would now be 01 May 2012 with a time of 00:00:00.

If you want to print the date to the console you can use several of its functions to print in different formats.

```
System.Console.WriteLine(t.ToString)
System.Console.WriteLine(t.ToShortDateString)
```

```
System.Console.WriteLine(t.ToString());
System.Console.WriteLine(t.ToShortDateString());
```

There are several other functions that can be looked up and used but generally I find these are the two that I use most often.

1.1.7 Doubles

Doubles are variables that hold numeric values with decimal places. They are similar to the decimal variable type but are less accurate and accumulate rounding errors when calculations are performed.

To declare a double you do this.

```
Dim d As Double
```

```
double d;
```

This example creates a new variable of type `Double` named `d`. It does not assign any value to `d`. To assign a value to `d` you can do it like this.

```
d = 5.555567
```

```
d = 5.555567;
```

Now the value of `d` is 5.555567.

```
d = 2.1
```

```
d = 2.1;
```

Now the value of `d` is 2.1.

Like any other variable type you can print the variable to the console using like this.

```
System.Console.WriteLine(d)
```

```
System.Console.WriteLine(d);
```

1.1.8 Objects

Objects are a base type that all other objects are derived from. This means that any other variable no matter the type can be assigned to an object.

To declare an object you do this.

```
Dim o As Object
```

```
Object o;
```

This example creates a new variable of type Object named o. It does not assign any value to o. To assign a value to o you can do it like this.

```
o = "A"c
```

```
o = 'A';
```

Now the value of o is A. The type is Char stored in the object. If we assign an integer.

```
o = 120
```

```
o = 120;
```

Now the value of o is 120 and the type of the stored value is an Integer

We can do the same by assigning strings, decimals, doubles, or any other type or object into an object of type Object.

If we print the object when assigned an integer it will print the integer. If we print when it is assigned char it will print the char and so on with the other variable types.

```
System.Console.WriteLine(o)
```

```
System.Console.WriteLine(o);
```

Generally I suggest avoiding the Object type as it defeats type checking that a compiler does and in my experience causes a lot of run time errors. The runtime errors are caused when code attempts to do an operation on the object that is not supported by the stored variable type. If we declare the type we want to use in code the compiler can do all the checks that are needed when the program is compiled.

1.2 Objects

VB and c# have built in types such as int, bool, string, and others. Now it is time to create types that is more specific to your application

For example if you are writing an application about tv channels/stations you probably do not want to use strings and integers. It will be easier to think about stations and channels. In VB, c#, and other object oriented languages we can define our own types and use them just like the built in types.

To use a class you must declare one as you would any other variable type. I will be using the word class and object interchangeably.

For example you declare an integer like this

```
Dim i As Integer = 0
```

```
int i = 0;
```

Below we create our own data type using the keyword Class. The best way to use a class is to think of it as an object. For the purpose of this example our object is going to be a tv show. Tv shows have many different aspects to them so we create an object that represent them.

Included in the class are a new class property (ShowName) as Public and a Private variable (_showName) that the property works with. Never declare a class variable as public. Always use a property or a function. I will not explain it here but I do encourage you to read some books on object oriented design.

Private variables are accessible from any function in the class but cannot be accessed from other classes.

```
Public Class TVShow
    Public Sub New()
```

```

        ' constructor
    End Sub

    Private _showName As String
    ' Public properties can be accessed from any function inside the
    ' class as well as other classes
    Public Property ShowName() As String
        Get
            ' Inside the get part the private variable is returned.
            ' You can do anything you want here such as data validation
            ' before returning the data if you need or want.
            Return _showName
        End Get
        Set(ByVal value As String)
            ' Inside the set part the private variable is set.
            ' You can do anything you want here such as data validation
            ' before the data is set.
            If value.Trim = "" Then
                Throw New Exception("ShowName cannot be empty")
            End If
            _showName = value
        End Set
    End Property

    ' The above property is long form. A shorter form can be done as seen below
    Public Property ShowLength As Integer
    Public Property Summary As String
    Public Property Rating As Decimal
    Public Property Episode As String
End Class

```

```

public class TVShow
{
    public TVShow()
    {
    }

    private string _showName;
    // Public properties can be accessed from any function inside the
    // class as well as other classes
    public string ShowName
    {
        get
        {
            // Inside the get part the private variable is returned.
            // You can do anything you want here such as data validation
            // before returning the data if you need or want.
            return _showName;
        }
        set
        {
            // Inside the set part the private variable is set.
            // You can do anything you want here such as data validation
            // before the data is set.

```

```

        if (value.Trim() == "")
            throw new Exception("ShowName cannot be empty");
        _showName = value;
    }
}

// The above property is long form. A shorter form can be done as seen below
public int ShowLength {get; init;}
public string Summary {get; init;}
public decimal Rating {get; init;}
public string Episode {get; init;}
}

```

You create a new instance of a class the same way you would with an Integer. You create a new instance like this

```

Dim starTrek As New TVShow With {
    .ShowName = "Star Trek",
    .ShowLength = 1380,
    .Summary = "Teleport Disaster",
    .Rating = 5.0D,
    .Episode = "1x12"
}

```

```

var starTrek = new TVShow() {
    ShowName = "Star Trek",
    ShowLength = 1380,
    Summary = "Teleport Disaster",
    Rating = 5.0m,
    Episode = "1x12"
};

```

If you want a second object you just declare another one.

```

Dim dexter As New TVShow With {
    .ShowName = "Dexter",
    .ShowLength = 1380,
    .Summary = "Dexter kills again.",
    .Rating = 4.8D,
    .Episode = "10x01"
}

```

```

var dexter = new TVShow() {
    ShowName = "Dexter",
    ShowLength = 1380,
    Summary = "Dexter kills again.",
    Rating = 4.8m,
    Episode = "10x01"
};

```

1.2.1 Methods

Methods, also known as functions, are used to break code apart into smaller chunks. Functions should do one task and do it well. Functions can be called again and again. They are used to keep duplicate code from building up. This makes things easier to understand. They can be chained/used together to perform complex tasks.

Functions can return a value or return no value. In vb functions that return a value use the key word **Function** and ones that do not return a value use the keyword **Sub**. In c# functions that return a value have a **type** such as a built-in type or object and functions that do not return a value use the keyword **void**.

```
public class TVShow
{
    public string ShowName {get; init;}
    public int ShowLength {get; init;}
    public string Summary {get; init;}
    public decimal Rating {get; init;}
    public string Episode {get; init;}

    // includeSummary is a method parameter
    public void PrettyPrint(bool includeSummary){
        if (includeSummary)
        {
            Console.WriteLine($"{ShowName} {Episode} {Rating} {ShowLength} {Summary}");
        }
        else
        {
            Console.WriteLine($"{ShowName} {Episode} {Rating} {ShowLength}");
        }
    }

    public bool IsGoodRating(){
        return Rating >= 3.0m;
    }
}

var dexter = new TVShow() {
    ShowName = "Dexter",
    ShowLength = 1380,
    Summary = "Dexter kills again.",
    Rating = 4.8m,
    Episode = "10x01"
};

dexter.PrettyPrint(includeSummary: true);

if(dexter.IsGoodRating()){
    Console.WriteLine("Let's watch this episode.");
}
```

Method and function parameters are passed by reference for objects and by value for simple types.

1.3 Interfaces

Interfaces - define behavior for multiple types. An interface contains definitions for a group of related functionalities that a non-abstract class or a struct must implement.

Interfaces in c# and vb is a way to specify what an object implements. It provides the ability to have different concrete class implementations and choose different ones at runtime.

A good example for further self study is the [Microsoft ILogger](#).

We will build upon the TVShows class. We will define an interface. We will include a new property ParentalGuide.

Much of this will not make sense until [Structuring an Application](#), which covers IOC and dependency injection.

The following code example defines an interface named TVShow. It is not necessary or necessarily recommended to prepend the name with an I but it is very common to see such interfaces in the c# and vb world. In code bases that do prepend an I the name would be ITVShow. The following code examples will not follow that pattern.

Imagine we have a large program involving tv shows. We could pass around the instances of TVShow but that will make our program brittle if and when we need to make changes.

```
public interface TVShow
{
    string ShowName {get; init;}
    int ShowLength {get; init;}
    string Summary {get; init;}
    decimal Rating {get; init;}
    string Episode {get; init;}
    string ParentalGuide {get; init;}

    void PrettyPrint(bool includeSummary);
    bool IsGoodRating();
}
```

An interface cannot be initialized. If we were to try to do so it would be a compile time error.

```
// Will not compile.
var inst = new TVShow();
```

Below a new class called ComedyShow implements TVShow. Notice line one with : **TVShow** after the class name. ComedyShow is a type of TVShow. Next notice that AdventureShow also implements TVShow.

```
public class ComedyShow : TVShow
{
    public string ShowName {get; init;}
    public int ShowLength {get; init;}
    public string Summary {get; init;}
    public decimal Rating {get; init;}
    public string Episode {get; init;}
    public string ParentalGuide {get; init;}

    // includeSummary is a method parameter
    public void PrettyPrint(bool includeSummary){
        if (includeSummary)
        {
            Console.WriteLine($"Comedy: {ShowName} {Episode} {Rating} {ShowLength} {Summary}");
        }
        else
        {
            Console.WriteLine($"Comedy: {ShowName} {Episode} {Rating} {ShowLength}");
        }
    }

    public bool IsGoodRating(){
```

```

        return Rating >= 3.0m;
    }
}

public class AdventureShow : TVShow
{
    public string ShowName {get; init;}
    public int ShowLength {get; init;}
    public string Summary {get; init;}
    public decimal Rating {get; init;}
    public string Episode {get; init;}
    public string ParentalGuide {get; init;}

    // includeSummary is a method parameter
    public void PrettyPrint(bool includeSummary){
        if (includeSummary)
        {
            Console.WriteLine($"Adventure: {ShowName} {Episode} {Rating} {ShowLength} {Summary}");
        }
        else
        {
            Console.WriteLine($"Adventure: {ShowName} {Episode} {Rating} {ShowLength}");
        }
    }

    public bool IsGoodRating(){
        return Rating >= 3.5m;
    }
}

```

Reviewing the code we can see that while both the ComedyShow and AdventureShow classes are similar they have different implementations of PrettyPrint and IsGoodRating. In addition to different internals to the interface methods they each could have different private helper methods or even other public methods.

Lets assume our application permits users to enter tv show information and as part of that entry they can add the show as a comedy or an adventure show. Let's store that information in a list. Notice how InsertShow has a parameter TVShow but lower in the code when calling the method all objects that implement the TVShow interface can be added and worked on.

```

public static class Shows
{
    static List<TVShow> _tvShows = new List<TVShow>();

    public static void InsertShow(TVShow show)
    {
        _tvShows.Add(show);
    }

    public static void PrintShows()
    {
        foreach (var show in _tvShows)
        {
            show.PrettyPrint(includeSummary: true);
        }
    }
}

```

```

    }
}

public static void Main()
{
    Shows.InsertShow(new ComedyShow() {
        ShowName = "Friends",
        ShowLength = 1380,
        Summary = "The friends get coffee.",
        Rating = 4.8m,
        Episode = "4x05",
        ParentalGuide = "PG13"
    });
    Shows.InsertShow(new AdventureShow() {
        ShowName = "Rick and morty",
        ShowLength = 760,
        Summary = "A quick 20 minute in and out adventure.",
        Rating = 3.8m,
        Episode = "3x14",
        ParentalGuide = "18A"
    });

    Shows.PrintShows();
}

```

The output is

```

Comedy: Friends 4x05 4.8 1380 The friends get coffee. Adventure: Rick and morty 3x14 3.8
760 A quick 20 minute in and out adventure.

```

For simplicity the example above is using a static Shows class. I almost always recommend against using static classes. I've shown their use in the above example as it is simple but in general I have found their use often coincides with global variables and long term they cause a maintenance quagmire. Static classes and variables have their place but try to avoid them.

Note: Read up about base classes and abstract bases classes as they are an alternative to using interfaces. Read about **SOLID** development.

Chapter 2

Asynchronous Work and Threads

Work that waits and work that computes are different problems, and .NET gives them different tools. Confusing the two is the usual source of both sluggish applications and mysterious data corruption.

Async and await are for waiting: a network call, a database query, a file read. The thread is released while the wait happens rather than sitting idle. Threads and the thread pool are for computing: work that genuinely needs a CPU. This chapter covers both, the locking that becomes necessary the moment two of them touch the same variable, and an in memory work queue for handing a slow job off and answering the caller straight away.

2.1 Async/Await

Asynchronous programming . The core of async programming is the Task and Task objects, which model asynchronous operations. They are supported by the async and await keywords. The model is fairly simple in most cases: For I/O-bound code, you await an operation that returns a Task or Task inside of an async method. For CPU-bound code, you await an operation that is started on a background thread with the Task.Run method.

Async and await provides a way for more efficient use of threads. When a task is run it can be awaited later while doing more work while waiting.

Simple async/await example:

```
Private Async Function LoadPreviousSettings() As Task
    Await Task.Delay(5000)
End Function
```

```
Dim loadTask As Task = LoadPreviousSettings()
```

```
' Do some other crazy stuff
```

```
Await loadTask
```

```
private async Task LoadPreviousSettings()
{
    await Task.Delay(5000);
}
```

```
var loadTask = LoadPreviousSettings();
```

```
// Do some other crazy stuff
```

```
await loadTask;
```

The async and await pattern makes asynchronous programming easier and feels more like sequential development. Good places for async/await is I/O bound work such as when making network calls. Much of the time is spent waiting for a response and the thread could be doing other work while waiting. Network calls such as database connections, commands, updates, inserts, selects, deletes, and stored procedure and functions executions should be run with async and await pattern.

Another place async/await should be used is when making http calls. The example below demonstrates using async/await when using HttpClient to download a web site front page. In an asp.net core application IHttpClientFactory should be used to create an HttpClient.

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Threading.Tasks;
using System.Net.Http;

public static async Task Main()
{
    // normally disposable objects should be disposed.
    // HttpClient is a special case and its norm is
    // that it should not be disposed until the program terminates
    using var client = new HttpClient();
    // add all tasks to a list and later await them.
    var tasks = new List<Task<string>>();
    Stopwatch stopWatch = new Stopwatch();
    stopWatch.Start();
    for(int i=0; i<10; i++)
    {
        var instDownloader = new Downloader();
        tasks.Add(instDownloader.DownloadSiteAsync(client, "https://majorsilence.com"));
    }
    Console.WriteLine("majorsilence.com is being downloaded 10 times. Waiting...");
    foreach(var t in tasks)
    {
        string html = await t;
        Console.WriteLine(html.Substring(0, 100));
    }
    stopWatch.Stop();
    TimeSpan ts = stopWatch.Elapsed;
    Console.WriteLine($"Code Downloaded in {ts.TotalMilliseconds} Milliseconds");

    // sequential async calls
    Console.WriteLine("start sequential async calls to download majorsilence.com. Waiting...");
    stopWatch.Restart();
    for(int i=0; i<10; i++)
    {
        var instDownloader = new Downloader();
        string html = await instDownloader.DownloadSiteAsync(client, "https://majorsilence.com");
        Console.WriteLine(html.Substring(0, 100));
    }
    stopWatch.Stop();
}
```

```

        TimeSpan ts2 = stopWatch.Elapsed;
        Console.WriteLine($"Sequential Code Downloaded in {ts2.TotalMilliseconds} Milliseconds");
    }

    public class Downloader{
        public async Task<string> DownloadSiteAsync(HttpClient httpClient,
            string url,
            System.Threading.CancellationToken cancellationToken = default(System.Threading.CancellationTok
        {
            var request = new HttpRequestMessage()
            {
                Method = HttpMethod.Get,
                RequestUri = new Uri(url)
            };

            // proceed past user agent sniffing
            request.Headers.Add("User-Agent", "Mozilla/5.0 (X11; CrOS x86_64 14541.0.0) AppleWebKit/537

            HttpResponseMessage response = await httpClient.SendAsync(request, cancellationToken).Confi

            response.EnsureSuccessStatusCode();
            return await response.Content.ReadAsStringAsync().ConfigureAwait(false);
        }
    }
}

```

2.2 Threads

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

Dot net provides the `Thread` class.

Here is an example that starts a background tasks and checks every 500 millisecond if it is complete using the `IsAlive` property. If the background thread is still working it continues its work inside a while loop.

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        var t = new Thread(ThreadMethod);
        t.Start();

        Console.WriteLine("Do other things while waiting for the background thread to finish");

        while(t.IsAlive){
            Console.WriteLine("Alive");
            await Task.Delay(500);
        }
    }
}

```

```

        Console.WriteLine("job completed");
    }

    static void ThreadMethod(){
        Console.WriteLine("The code in this method is running in its own thread.");
        Console.WriteLine("Sleep the thread 5000 milliseconds to demonstrate the main thread keeps working");
        Thread.Sleep(5000);
    }
}

```

This example starts a thread and does no work. The main thread stops work and waits for the background thread to complete using the Join method.

```

using System;
using System.Collections.Generic;
using System.Threading;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        var t = new Thread(ThreadMethod);
        t.Start();
        Console.WriteLine("Wait for the background thread to complete");
        t.Join();
        Console.WriteLine("job completed");
    }

    static void ThreadMethod(){
        Console.WriteLine("The code in this method is running in its own thread.");
        for(int i = 1; i < 6; i++){
            Console.WriteLine($"background loop count {i}");
            Thread.Sleep(500);
        }
    }
}

```

2.2.1 Locks

If more than one thread or task is updating a variable you should lock the variable as necessary.

The example below create multiple tasks that all update the same “count” variable. As you can see it locks the variable before updating it.

```

Dim tasks As New List(Of Task)
Dim lockObject As New Object()

Dim count As Integer = 0

For i As Integer = 0 To 9
    tasks.Add(Task.Factory.StartNew(Sub()
        For j As Integer = 0 To 999
            SyncLock lockObject
                count = count + 1
            End SyncLock
        Next j
    End Sub))
Next i

```

```

        End SyncLock
    Next
End Sub))
Next

For Each t In tasks
    Await t
Next

System.Console.WriteLine(count)

using System;
using System.Collections.Generic;
using System.Threading.Tasks;

public class Program
{
    public static async Task Main()
    {
        var tasks = new List<Task>();
        var lockObject = new object();

        int count = 0;
        for (int i = 0; i < 10; i++)
        {
            tasks.Add(Task.Factory.StartNew(() =>
            {
                for (int j = 0; j <= 999; j++)
                {
                    lock (lockObject)
                    {
                        count = count + 1;
                    }
                }
            }));
        }

        foreach (var t in tasks)
        {
            await t;
        }

        Console.WriteLine(count);
    }
}

```

On .NET 9 and newer prefer the dedicated `System.Threading.Lock` type over locking on a plain `object`. The c# compiler recognises it and emits the faster `Lock.EnterScope` path. The only change needed is the declaration.

```

// was: var lockObject = new object();
var lockObject = new System.Threading.Lock();

lock (lockObject)
{

```

```
count = count + 1;
}
```

For the specific case of incrementing a counter, skip the lock entirely and use `Interlocked`, which is cheaper.

```
System.Threading.Interlocked.Increment(ref count);
```

2.3 In Memory Work Queue

Sometimes work should be accepted now and performed later, without involving Redis or any other external broker. A web request that triggers a slow report, or a desktop app that queues uploads, both want the same thing: hand the item off, return immediately, and let a background worker drain the queue.

The two approaches below are both in process. If the application restarts, anything still queued is lost. When that is unacceptable, use a durable queue such as the Redis one shown in the **Work and Message Queue with Redis** section, or a library such as `Hangfire`.

2.3.1 Task Queue

`System.Threading.Channels` is the modern way to do this. A channel is a thread safe producer/consumer queue that supports async reads, so a consumer waits without burning a thread.

Set a bounded capacity. An unbounded queue will happily grow until the process runs out of memory when producers outpace the consumer.

```
using System;
using System.Threading;
using System.Threading.Channels;
using System.Threading.Tasks;

public class WorkQueue
{
    private readonly Channel<Func<CancellationToken, Task>> _channel =
        Channel.CreateBounded<Func<CancellationToken, Task>>(
            new BoundedChannelOptions(capacity: 100)
            {
                // block the producer rather than dropping work
                FullMode = BoundedChannelFullMode.Wait
            }
        );

    public async Task EnqueueAsync(Func<CancellationToken, Task> workItem)
    {
        ArgumentNullException.ThrowIfNull(workItem);
        await _channel.Writer.WriteAsync(workItem);
    }

    public IAsyncEnumerable<Func<CancellationToken, Task>> ReadAllAsync(
        CancellationToken cancellationToken) =>
        _channel.Reader.ReadAllAsync(cancellationToken);
}
```

The consumer loop. In asp.net core this belongs in a `BackgroundService`, registered with `services.AddHostedService<QueueWorker>()` and the queue itself as a singleton.

```
public class QueueWorker : BackgroundService
{
    ...
}
```

```

private readonly WorkQueue _queue;
private readonly ILogger<QueueWorker> _logger;

public QueueWorker(WorkQueue queue, ILogger<QueueWorker> logger)
{
    _queue = queue;
    _logger = logger;
}

protected override async Task ExecuteAsync(CancellationToken stoppingToken)
{
    await foreach (var workItem in _queue.ReadAllAsync(stoppingToken))
    {
        try
        {
            await workItem(stoppingToken);
        }
        catch (Exception ex)
        {
            // one bad work item must not kill the worker
            _logger.LogError(ex, "Work item failed");
        }
    }
}

```

Queueing work from a controller then costs one line, and the caller gets its response straight away.

```

await _queue.EnqueueAsync(async token =>
{
    await _reportBuilder.GenerateAsync(reportId, token);
});

```

The try/catch around `workItem` is the part people leave out. Without it, one unhandled exception ends the `await foreach` loop and the queue silently stops draining for the rest of the process lifetime.

2.3.2 Thread Queue

If the work is CPU bound rather than I/O bound, running it on the thread pool is usually all that is required. `Task.Run` queues the delegate to the pool, which already manages a pool of threads for you.

```

var work = Task.Run(() => ExpensiveCalculation(input));

// do other things

int result = await work;

```

For CPU bound work over a collection, `Parallel.ForEachAsync` limits how many run at once, which avoids swamping the pool.

```

await Parallel.ForEachAsync(
    shows,
    new ParallelOptions { MaxDegreeOfParallelism = Environment.ProcessorCount },
    async (show, token) =>
    {
        await ProcessShowAsync(show, token);
    }
);

```

```
});
```

Dedicated `Thread` objects, as shown in the **Threads** section, are worth the trouble only for long running work that should not occupy a pool thread for minutes at a time. In that case set `IsBackground = true` so the thread does not keep the process alive at shutdown.

```
var t = new Thread(ThreadMethod) { IsBackground = true };  
t.Start();
```

Chapter 3

Structuring an Application

The previous chapters were about writing code that works. This one is about arranging it so that it keeps working after a year of changes and can be tested without standing up a database.

The three patterns here reinforce each other. Inversion of control means a class is handed what it depends on rather than constructing it. The repository pattern puts data access behind an interface, which is what gives inversion of control something worth injecting. Events let one part of an application tell others that something happened without knowing who is listening. The chapter closes on packaging code as NuGet, and on testing, which is the payoff for all of it.

3.1 IOC

Inversion of Control (IOC) means a class does not create the things it depends on. It is given them instead. **Dependency injection (DI)** is the usual way to do that: dependencies are passed into the constructor, and something else decides which concrete implementation to supply.

This is what makes the interfaces and repository classes shown in the previous sections worth writing. `TestStuff` takes an `ITestRepo` and never learns whether it is talking to SQL Server, SQLite, or a mock in a unit test.

.NET ships a DI container in `Microsoft.Extensions.DependencyInjection`. In an asp.net core app it is already wired up. For a console app or a Winforms app, add the package.

```
dotnet add package Microsoft.Extensions.DependencyInjection
```

3.1.1 Registering and resolving services

Registration maps an interface to the concrete class that implements it. Resolution walks the constructor parameters and builds the whole object graph for you.

```
using System;
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

// map the interface to the implementation
services.AddSingleton<ITestRepo>(sp =>
    new MajorSilence.DataAccess.TestRepo("Data Source=shows.db"));
services.AddTransient<MajorSilence.BusinessStuff.TestStuff>();

using var provider = services.BuildServiceProvider();
```

```
// TestStuff needs an ITestRepo. The container supplies it.
var inst = provider.GetRequiredService<MajorSilence.BusinessStuff.TestStuff>();
inst.DoStuff();
```

Compare that with the manual wiring in the **Repository Pattern** section. For two classes the manual version is fine. Once an application has fifty of them, the container is what keeps **Main** from becoming a wall of **new**.

3.1.2 Lifetimes

Choosing the wrong lifetime is the most common source of DI bugs, so it is worth being deliberate about it.

- **Transient** - a new instance every time it is requested. The safe default for cheap, stateless classes.
- **Scoped** - one instance per scope. In asp.net core a scope is one http request. This is what **DbContext** should use.
- **Singleton** - one instance for the life of the application. Must be thread safe, since many threads can use it at once.

The rule that catches people out: a singleton must never depend on a scoped service. The singleton is built once and holds onto whatever it was given, so it would keep using the first request's scoped instance forever. The container will throw at startup if you try, provided scope validation is on, which it is by default in development.

3.1.3 Registering by convention

Registering interfaces one by one is tedious in a large solution. **Scrutor** scans an assembly and registers everything matching a convention.

```
dotnet add package Scrutor
```

```
services.Scan(scan => scan
    .FromAssemblyOf<MajorSilence.DataAccess.ITestRepo>()
    .AddClasses(classes => classes.Where(t => t.Name.EndsWith("Repo")))
    .AsImplementedInterfaces()
    .WithScopedLifetime());
```

3.1.4 Why this makes testing easy

Because **TestStuff** only ever sees **ITestRepo**, a test can hand it a stand-in and assert on what it did, with no database involved. See **mocking**.

```
using Moq;
using NUnit.Framework;

[Test]
public void DoStuffInsertsTheName()
{
    var repo = new Mock<ITestRepo>();
    repo.Setup(x => x.GetName()).Returns("The Name");

    var inst = new MajorSilence.BusinessStuff.TestStuff(repo.Object);
    inst.DoStuff();

    repo.Verify(x => x.InsertData("The Name"), Times.Once);
}
```

3.1.5 Other containers

The built in container covers most needs. Third party containers add features such as property injection, interception, and more advanced conditional registration.

- [Autofac](#)
- [Lamar](#)

3.2 Repository Pattern

Use the repository pattern to separate your business and data access layers. Makes it easy to test your business and data layer code separately.

There are different ways to do this. Here are a couple ways.

3.2.1 Use a base abstract class that is passed a connection

```
using System;
using System.Data;
using Microsoft.Data.Sqlite;
using System.Threading.Tasks;

namespace MajorSilence.DataAccess
{
    public abstract class BaseRepo
    {
        private readonly string cnStr;

        protected BaseRepo(string cnStr)
        {
            this.cnStr = cnStr;
        }

        protected T WithConnection<T>(Func<IDbConnection, T> sqlTransaction)
        {
            using (var connection = new SqliteConnection(cnStr))
            {
                connection.Open();
                return sqlTransaction(connection);
            }
        }

        protected void WithConnection(Action<IDbConnection> sqlTransaction)
        {
            using (var connection = new SqliteConnection(cnStr))
            {
                connection.Open();
                sqlTransaction(connection);
            }
        }

        protected async Task<T> WithConnectionAsync<T>(Func<IDbConnection, Task<T>> sqlTransaction)
        {
            using (var connection = new SqliteConnection(cnStr))
            {

```

```

        await connection.OpenAsync();
        return await sqlTransaction(connection);
    }
}

protected async Task WithConnectionAsync(Func<IDbConnection, Task> sqlTransaction)
{
    using (var connection = new SqliteConnection(cnStr))
    {
        await connection.OpenAsync();
        await sqlTransaction(connection);
    }
}
}
}
}

```

And here is the repo class

```

using System;
using System.Linq;
using Dapper;

namespace MajorSilence.DataAccess
{
    public interface ITestRepo
    {
        string GetName();
        void InsertData(string name);
    }

    public class TestRepo : BaseRepo, ITestRepo
    {
        public TestRepo(string cnStr) : base(cnStr) { }

        public string GetName()
        {
            return this.WithConnection(cn =>
            {
                return cn.Query<string>("SELECT Name From TheTable LIMIT 1;").FirstOrDefault();
            });
        }

        public void InsertData(string name)
        {
            this.WithConnection(cn =>
            {
                cn.Execute("INSERT INTO TheTable (Name) VALUES (@Name);",
                    new { Name = name });
            });
        }
    }
}

```

3.2.2 No base abstract. Let individual repository classes do as they please

I generally prefer this way. It is simple.

```
using Microsoft.Data.Sqlite;
using System.Linq;
using Dapper;

namespace MajorSilence.DataAccess
{
    public class TestRepoNobase : ITestRepo
    {
        readonly string cnStr;
        public TestRepoNobase(string cnStr)
        {
            this.cnStr = cnStr;
        }

        public string GetName()
        {
            using (var cn = new SqliteConnection(cnStr))
            {
                return cn.Query<string>("SELECT Name From TheTable LIMIT 1;").FirstOrDefault();
            }
        }

        public void InsertData(string name)
        {
            using (var cn = new SqliteConnection(cnStr))
            {
                cn.Execute("INSERT INTO TheTable (Name) VALUES (@Name);",
                    new { Name = name });
            }
        }
    }
}
```

3.2.3 Do something with the repository classes

A business class

```
using System;
namespace MajorSilence.BusinessStuff
{
    public class TestStuff
    {
        readonly DataAccess.ITestRepo repo;
        public TestStuff(DataAccess.ITestRepo repo)
        {
            this.repo = repo;
        }

        public void DoStuff()
        {
            repo.InsertData("The Name");
        }
    }
}
```

```

        string name = repo.GetName();

        // Do stuff with the name
    }
}

```

Combine everything. Manually initialize our two repository classes and initialize two copies of our TestStuff class. Our TestStuff never knows what or where the actual data layer is.

Note the file backed connection string. Both repository classes open a fresh connection per call, and with SQLite a plain `Data Source=:memory:` gives every connection its own private, empty database - so the insert and the select would not see each other. If you want an in memory database that survives across connections, use `Data Source=shows;Mode=Memory;Cache=Shared` and keep one connection open for as long as the database should live.

TestStuff is now easily tested with tools such as `moq`.

```

using System;

namespace MajorSilence.TestStuff
{
    class Program
    {
        static void Main(string[] args)
        {
            // Our repository layer that will talk to the data source.
            // This could be inject with a dependency injection framework
            var repo = new MajorSilence.DataAccess.TestRepo("Data Source=shows.db");
            var repo2 = new MajorSilence.DataAccess.TestRepoNobase("Data Source=shows.db");

            // Our business class. Takes an interface and does not care
            // what the actual data source is.
            var inst = new MajorSilence.BusinessStuff.TestStuff(repo);
            inst.DoStuff();

            var inst2 = new MajorSilence.BusinessStuff.TestStuff(repo2);
            inst2.DoStuff();
        }
    }
}

```

3.3 Events

Custom Event and Event Handlers

3.3.1 Use built in EventHandler

```

public class TheExample
{
    public event System.EventHandler DoSomething;
}

```

```

public void TheTest(){
    // option 1 to raise event
    this.DoSomething?.Invoke(this, new System.EventArgs());

    // option 2 to raise event
    if (DoSomething != null)
    {
        DoSomething(this, new System.EventArgs());
    }
}
}

```

3.3.2 Use custom delegate as event handler

```

public class TheExample
{
    public delegate void MyCustomEventHandler(object sender, System.EventArgs e);
    public event MyCustomEventHandler DoSomething;

    public void TheTest(){
        // option 1 to raise event
        this.DoSomething?.Invoke(this, new System.EventArgs());

        // option 2 to raise event
        if (DoSomething != null)
        {
            DoSomething(this, new System.EventArgs());
        }
    }
}

```

3.3.3 Subscribe to the event

```

// subscribe using lambda expression

var x = new TheExample();

x.DoSomething += (s,e) => {
    Console.WriteLine("hi, the event has been raised");
};

x.TheTest();

```

3.3.4 VB example of basic custom events

```

Public Class TheExample
    Public Delegate Sub MyCustomEventHandler(ByVal sender As Object, ByVal e As System.EventArgs)
    Public Event DoSomething As MyCustomEventHandler

    Public Sub TheTest()
        RaiseEvent DoSomething(Me, New EventArgs())
    End Sub
End Class

```

Subscribe to the event

```
Dim x As New TheExample
AddHandler x.DoSomething, AddressOf EventCallback
x.TheTest()
```

```
RemoveHandler x.DoSomething, AddressOf EventCallback
```

```
Sub EventCallback(ByVal sender As Object, ByVal e As System.EventArgs)
    Console.WriteLine("Hi, the event has been raised")
End Sub
```

3.3.5 Create a custom event

Setup a new custom event class inheriting from EventArgs and setup a new delegate.

```
public delegate void MyCustomEventHandler(object sender, MyCustomEvent e);

public class MyCustomEvent : System.EventArgs
{
    private string _msg;
    private float _value;

    public MyCustomEvent(string m)
    {
        _msg = m;
        _value = 0;
    }

    public MyCustomEvent(float v)
    {
        _msg = "";
        _value = v;
    }

    public string Message
    {
        get { return _msg; }
    }

    public float Value
    {
        get { return _value; }
    }
}
```

3.3.6 Use the custom event

```
public event MyCustomEventHandler DoSomething;

this.DoSomething?.Invoke(this, new MyCustomEvent(123.95f));
```

3.4 Nuget

Generally using nuget is very simple. Using Visual Studio right click your solution or project and select “Add Nuget Package”. Find your package and add it. It is auto added. Any time you now clone your project on a new computer the first time you build your project it will restore your nuget references.

3.4.1 Create a NuGet Package

Given a .csproj or .vbproj file with a PropertyGroup like the following, add the **GeneratePackageOnBuild**, **PackageProjectUrl**, **Description**, **Authors**, **RepositoryUrl**, **PackageLicenseExpression**, **Version**.

```
<PropertyGroup>
  <TargetFrameworks>netstandard2.0;net10.0</TargetFrameworks>
</PropertyGroup>
```

PropertyGroup that generates a nuget package on build and fills in many useful details.

```
<PropertyGroup>
  <TargetFrameworks>netstandard2.0;net10.0</TargetFrameworks>
  <GeneratePackageOnBuild>true</GeneratePackageOnBuild>
  <PackageProjectUrl>https://PLACEHOLDER</PackageProjectUrl>
  <Description>PLACEHOLDER</Description>
  <Authors>PLACEHOLDER</Authors>
  <RepositoryUrl>https://PLACEHOLDER</RepositoryUrl>
  <PackageLicenseExpression>MIT</PackageLicenseExpression>
  <Version>1.0.1</Version>
</PropertyGroup>
```

3.4.2 Add NuGet source

The following command will add a nuget source to your computer other than the default. This is good for self hosted nuget servers. Use `--store-password-in-clear-text` if a mac or linux workstation is being used.

```
dotnet nuget add source "https://your.source.url/v3/index.json" -n [Feed Name] -u YourUserName -p Y
```

3.4.3 Check if NuGet Source already Exists

The following powershell script will check if a nuget source already exists on your computer.

```
dotnet nuget list source
```

3.4.4 Start fresh with just nuget.org

```
dotnet new nugetconfig
```

3.5 Testing and Coverage

3.5.1 NUnit and Coverlet

NUnit is a fine testing framework for c#, vb and other .net based languages.

The nuget package **NUnit** must be referenced for base NUnit support in a test project, and **NUnit3TestAdapter** is what actually lets the test runner find the tests - leave it out and `dotnet test` reports zero tests without ever saying why. Despite the 3 in its name it is the current adapter for NUnit 4 as well. **NunitXml.TestLogger** produces NUnit format result files for CI systems that expect them.

For integration within visual studio and rider **Microsoft.NET.Test.Sdk** should also be added to the test project. **coverlet.collector** is used to generate the code coverage report. Note, for large solutions and projects coverlet can add a considerable overhead.

```
dotnet add package NUnit
dotnet add package NUnit3TestAdapter
dotnet add package NunitXml.TestLogger
dotnet add package Microsoft.NET.Test.Sdk
dotnet add package coverlet.collector
```

To demonstrate the the nunit testing framework we will work with a contrived example. The test class will test a modified threaded lock example from above.

Within the test class **ComplexAdditionTests** the code will confirm that the calculation works. This is helpful if a developer ever changes the `CalculateWithLock` method and breaks it. The test will fail and the developer will know that the change causes problems. The test will test the class **ComplexAddition**.

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using NUnit.Framework;

[TestFixture]
public class ComplexAdditionTests
{
    [Test]
    public async Task CalculationsCalculatesTest()
    {
        var complexAdds = new ComplexAddition();

        // 10 outer iterations, each adding 1 once per inner value 0..999
        const int expectedResult = 10 * 1000;
        int actualResult = await complexAdds.CalculateWithLock(10, 999);

        Assert.That(actualResult, Is.EqualTo(expectedResult));
    }
}

public class ComplexAddition
{
    public async Task<int> CalculateWithLock(int outerLimit = 10, int innerLimit = 999)
    {
        var tasks = new List<Task>();
        var lockObject = new object();

        int count = 0;
        for (int i = 0; i < outerLimit; i++)
        {
            tasks.Add(Task.Factory.StartNew(() =>
            {
                for (int j = 0; j <= innerLimit; j++)
                {
                    lock (lockObject)
                    {
                        count = count + 1;
                    }
                }
            }
            ));
        }

        await Task.WhenAll(tasks);

        return count;
    }
}
```

```
        }
    }));
}

foreach(var t in tasks)
{
    await t;
}

return count;
}
}
```

The tests can be run from within visual studios test explorer or from the command line with either **dotnet test**.

```
dotnet test
```

To test and collect coverage data run dotnet test with collector arguments.

```
dotnet test -c Release YourSolutionFile.sln --collect:"XPlat Code Coverage" --logger:"nunit"
```

Passing extra args example with exclude by file.

```
dotnet test -c Release YourSolutionFile.sln --collect:"XPlat Code Coverage" --logger:"nunit" -- Dat
```

3.5.2 Other test frameworks

Unit test frameworks:

- xUnit
- [MSTest](#)

Acceptance testing framework

- [FitNesse](#)

BDD (Behavior-driven development) testing

- [Reqnroll](#)

Part II

User Interfaces

Chapter 4

Desktop User Interfaces

Desktop applications did not go away, and .NET still has good answers for them.

Windows Forms is the older of the two covered here and the more direct: a form, a designer, controls with events. It remains the fastest way to put a working tool in front of somebody, and via [Majorsilence.Forms](#) the same code can now run beyond Windows. .NET MAUI is the newer, cross platform framework, with one project targeting Windows, mac, Android and iOS, and XAML rather than a Windows only designer. Which to reach for depends mostly on whether you need to leave Windows and whether the application already exists.

4.1 Winforms

Windows Forms is the desktop UI framework that has shipped with .NET since the beginning. It is still supported and still a reasonable choice for line of business applications on Windows, especially when a team already knows it. On modern .NET it is Windows only, so set the target framework accordingly.

```
<PropertyGroup>
  <OutputType>WinExe</OutputType>
  <TargetFramework>net10.0-windows</TargetFramework>
  <UseWindowsForms>true</UseWindowsForms>
  <Nullable>enable</Nullable>
</PropertyGroup>
```

Create a new project from the command line.

```
dotnet new winforms -o YourApp
cd YourApp
dotnet run
```

4.1.1 A form with a control and an event handler

The designer generates most of this for you, but it is useful to see what it produces. A form is a class that inherits from `Form`, controls are fields on that class, and user interaction is handled by subscribing to events.

```
using System;
using System.Windows.Forms;

public class ShowForm : Form
{
    private readonly TextBox _showName = new TextBox { Left = 10, Top = 10, Width = 200 };
}
```

```

private readonly Button _save = new Button { Left = 220, Top = 10, Text = "Save" };
private readonly ListBox _shows = new ListBox { Left = 10, Top = 45, Width = 410, Height = 200 };

public ShowForm()
{
    Text = "TV Shows";
    ClientSize = new System.Drawing.Size(440, 260);
    Controls.AddRange(new Control[] { _showName, _save, _shows });

    _save.Click += Save_Click;
}

private void Save_Click(object sender, EventArgs e)
{
    if (string.IsNullOrEmpty(_showName.Text))
    {
        MessageBox.Show("Show name cannot be empty", "Validation");
        return;
    }

    _shows.Items.Add(_showName.Text);
    _showName.Clear();
}
}

```

4.1.2 Keep the UI thread responsive

Everything in the paragraphs above about `async/await` applies here, and it matters more in a desktop app than anywhere else. Any slow work performed directly in an event handler freezes the window, because the same thread that runs your handler also paints the form and processes input.

Mark the handler `async` and await the slow work. Winforms installs a synchronization context, so execution resumes on the UI thread after the `await` and it is safe to touch controls again. Do not use `ConfigureAwait(false)` in code that will go on to update the UI.

```

private async void Load_Click(object sender, EventArgs e)
{
    _load.Enabled = false;
    try
    {
        // runs off the UI thread, window stays responsive
        var shows = await _repo.GetShowsAsync();

        // back on the UI thread here
        _shows.Items.Clear();
        _shows.Items.AddRange(shows.ToArray());
    }
    finally
    {
        _load.Enabled = true;
    }
}

```

`async void` is normally something to avoid, but event handlers are the one place it is correct, because the event signature returns `void`. Wrap the body in a `try/catch` or an unhandled exception will take down

the process.

If you have work running on a background thread that is not awaited, you cannot update controls from it directly. Marshal back to the UI thread with `Invoke`.

```
if (_shows.InvokeRequired)
{
    _shows.Invoke(() => _shows.Items.Add(name));
}
else
{
    _shows.Items.Add(name);
}
```

4.1.3 Cross platform alternatives

Winforms does not run on linux or mac. If that matters:

- [Majorsilence.Forms](#) - a Winforms compatibility layer, useful for porting an existing Winforms codebase.
- [Avalonia](#) - a mature cross platform XAML UI framework.
- [.NET MAUI](#) - Microsoft's cross platform framework, see [Microsoft Maui](#) below.

4.2 Microsoft Maui

[.NET MAUI](#) (Multi-platform App UI) builds native desktop and mobile applications from a single c# codebase, targeting android, iOS, mac, and windows. It is the successor to `Xamarin.Forms`.

Unlike [Winforms](#) above, which is windows only, MAUI renders through each platform's own native controls, so an app looks like an android app on android and a mac app on mac.

Install the workload and create a project.

```
dotnet workload install maui
dotnet new maui -o YourApp
cd YourApp

# run on a specific platform
dotnet build -t:Run -f net10.0-android
dotnet build -t:Run -f net10.0-windows10.0.19041.0
```

A MAUI project targets several frameworks at once from one csproj.

```
<PropertyGroup>
  <TargetFrameworks>net10.0-android;net10.0-ios;net10.0-maccatalyst</TargetFrameworks>
  <TargetFrameworks Condition="$([MSBuild]::IsOSPlatform('windows'))">
    $(TargetFrameworks);net10.0-windows10.0.19041.0
  </TargetFrameworks>
  <OutputType>Exe</OutputType>
  <UseMaui>true</UseMaui>
  <SingleProject>true</SingleProject>
</PropertyGroup>
```

Building for iOS or mac requires a mac. Android and windows build anywhere.

4.2.1 Pages and XAML

UI is normally written in XAML, with the logic in a matching code behind file.

```
<?xml version="1.0" encoding="utf-8" ?>
<ContentPage xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
    xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
    x:Class="YourApp.ShowsPage"
    Title="TV Shows">
    <VerticalStackLayout Padding="20" Spacing="10">
        <Entry x:Name="ShowNameEntry" Placeholder="Show name" />
        <Button Text="Add" Clicked="OnAddClicked" />
        <CollectionView x:Name="ShowsList">
            <CollectionView.ItemTemplate>
                <DataTemplate>
                    <Label Text="{Binding ShowName}" FontSize="18" />
                </DataTemplate>
            </CollectionView.ItemTemplate>
        </CollectionView>
    </VerticalStackLayout>
</ContentPage>
```

```
public partial class ShowsPage : ContentPage
{
    private readonly ObservableCollection<TvShow> _shows = new();

    public ShowsPage()
    {
        InitializeComponent();
        ShowsList.ItemsSource = _shows;
    }

    private void OnAddClicked(object sender, EventArgs e)
    {
        if (string.IsNullOrWhiteSpace>ShowNameEntry.Text))
        {
            return;
        }

        _shows.Add(new TvShow { ShowName = ShowNameEntry.Text });
        ShowNameEntry.Text = "";
    }
}
```

`ObservableCollection<T>` is what makes the list update itself. It raises a change notification on add and remove, which the `CollectionView` listens for. A plain `List<T>` will not refresh the UI.

4.2.2 Dependency injection

MAUI uses the same container described in the **IOC** section. Register services in `MauiProgram.cs` and pages resolve their dependencies through the constructor.

```
public static class MauiProgram
{
    public static MauiApp CreateMauiApp()
    {
        var builder = MauiApp.CreateBuilder();
        builder.UseMauiApp<App>();
    }
}
```

```
builder.Services.AddSingleton<ITestRepo>(sp =>
    new TestRepoNobase("Data Source=shows.db"));
builder.Services.AddTransient<ShowsPage>();

return builder.Build();
}
```

4.2.3 Keeping the UI responsive

The same rule as Winforms applies: slow work in a handler freezes the UI. Await it instead, and marshal back to the UI thread from any background work using `MainThread`.

```
private async void OnLoadClicked(object sender, EventArgs e)
{
    var shows = await _repo.GetShowsAsync();

    MainThread.BeginInvokeOnMainThread(() =>
    {
        _shows.Clear();
        foreach (var show in shows)
        {
            _shows.Add(show);
        }
    });
}
```

4.2.4 Alternatives

- [Avalonia](#) - cross platform XAML UI, also runs on linux, which MAUI does not target.
- [Uno Platform](#) - WinUI style markup across mobile, desktop, and WebAssembly.

Chapter 5

Crystal Reports

Examples to use crystal reports from c#. Crystal reports for .net currently only supports running on .net framework 4.8 and older. If reports need to be generated on modern .net see [CrystalCmd Server and Client](#) below.

Make sure you have the crystal reports runtime installed. It can be downloaded from <https://wiki.scn.sap.com/wiki/display/BOBJ/Crystal+Reports%2C+Developer+for+Visual+Studio+Downloads>.

All examples below require references for **CrystalDecisions.CrystalReports.Engine** and **CrystalDecisions.Shared** to be added to your project.

Ensure the CrystalReports Version and PublicKey token match the installed version of Crystal Reports.

```
<ItemGroup>
  <Reference Include="CrystalDecisions.Windows.Forms, Version=13.0.4000.0, Culture=neutral, Publi
    <HintPath>C:\Windows\Microsoft.NET\assembly\GAC_MSIL\CrystalDecisions.Windows.Forms\v4.0_13
  </Reference>
  <Reference Include="CrystalDecisions.CrystalReports.Engine, Version=13.0.4000.0, Culture=neutra
    <HintPath>C:\Windows\Microsoft.NET\assembly\GAC_MSIL\CrystalDecisions.CrystalReports.Engine
  </Reference>
  <Reference Include="CrystalDecisions.Shared, Version=13.0.4000.0, Culture=neutral, PublicKeyTok
    <HintPath>C:\Windows\Microsoft.NET\assembly\GAC_MSIL\CrystalDecisions.Shared\v4.0_13.0.4000
  </Reference>
</ItemGroup>
```

5.1 Set data using DataTables

Example initializing a report and passing in a DataTable.

```
using CrystalDecisions.CrystalReports.Engine;
using CrystalDecisions.Shared;

// Pass a DataTable to a crystal report table
public static void SetData(string crystalTemplateFilePath,
    string tableName, DataSet val)
{
    using (var rpt = new ReportDocument())
    {
        rpt.Load(crystalTemplateFilePath);
    }
}
```

```

        rpt.Database.Tables[tableName].SetDataSource(val);
    }
}

// Pass any generic IEnumerable data to a crystal report DataTable
public static void SetData<T>(string crystalTemplateFilePath,
    string tableName, IEnumerable<T> val)
{
    using (var rpt = new ReportDocument())
    {
        rpt.Load(crystalTemplateFilePath);

        var dt = ConvertGenericListToDatatable(val);
        rpt.Database.Tables[tableName].SetDataSource(dt);
    }
}

// Found somewhere on the internet. I know longer remember where.
public static DataTable ConvertGenericListToDatatable<T>(IEnumerable<T> dataLst)
{
    DataTable dt = new DataTable();

    foreach (var info in dataLst.FirstOrDefault().GetType().GetProperties())
    {
        dt.Columns.Add(info.Name, info.PropertyType);
    }

    foreach (var tp in dataLst)
    {
        DataRow row = dt.NewRow();
        foreach (var info in typeof(T).GetProperties())
        {
            if (info.Name == "Item") continue;
            row[info.Name] = info.GetValue(tp, null) == null ? DBNull.Value : info.GetValue(tp, null);
        }
        dt.Rows.Add(row);
    }
    dt.AcceptChanges();
    return dt;
}

```

5.2 Set report parameters

Set report parameters from code.

```

using CrystalDecisions.CrystalReports.Engine;
using CrystalDecisions.Shared;

public static void SetParameterValueName(string crystalTemplateFilePath, object val)
{
    using (var rpt = new ReportDocument())
    {
        rpt.Load(crystalTemplateFilePath);
    }
}

```

```

        string name = "ParameterName";
        if (rpt.ParameterFields[name] != null)
        {
            rpt.SetParameterValue(name, val);
        }
    }
}

```

5.3 Move report objects

Example moving an object.

```

using CrystalDecisions.CrystalReports.Engine;
using CrystalDecisions.Shared;

public static void MoveObject(string crystalTemplateFilePath)
{
    using (var rpt = new ReportDocument())
    {
        rpt.Load(crystalTemplateFilePath);

        rpt.ReportDefinition.ReportObjects["objectName"].Left = 15;
        rpt.ReportDefinition.ReportObjects["objectName"].Top = 15;
    }
}

```

5.4 Export to pdf or other file type

Load a report and export it to pdf. You can pass in data or set other properties before the export.

```

using CrystalDecisions.CrystalReports.Engine;
using CrystalDecisions.Shared;

public static void ExportPdf(string crystalTemplateFilePath,
    string pdfFilename)
{
    using (var rpt = new ReportDocument())
    {
        rpt.Load(crystalTemplateFilePath);

        var exp = ExportFormatType.PortableDocFormat;
        rpt.ExportToDisk(exp, pdfFilename);
    }
}

```

5.5 CrystalCmd Server and Client

crystalcmd is a:

Java and c# program to load json files into crystal reports and produce PDFs.

- <https://github.com/majorsilence/CrystalCmd>

tldr: use crystal reports with dotnet netstandard2.0, net48, net8.0, net9.0, net10.0 on linux, windows, mac, android, and iOS.

To host the crystalcmd .net server browse to <https://github.com/majorsilence/CrystalCmd/tree/main/dotnet> and build the **Dockerfile.wine** and **Dockerfile.crystalcmd**. If a java server is required use the prebuilt image at <https://hub.docker.com/r/majorsilence/crystalcmd>. The c# server is recommended.

With a crystalcmd server running crystal report templates and data can be sent to it to produce pdf files. The docker images can run on any system that supports docker such as mac, windows, and linux.

Add the package **Majorsilence.CrystalCmd.Client** to your project.

To call a crystalcmd server use the nuget package **Majorsilence.CrystalCmd.Client**.

```
dotnet add package Majorsilence.CrystalCmd.Client
```

This example will call the server and return the pdf report as a stream.

```
DataTable dt = new DataTable();

// init report data
var reportData = new Majorsilence.CrystalCmd.Common.Data()
{
    DataTables = new Dictionary<string, string>(),
    MoveObjectPosition = new List<Majorsilence.CrystalCmd.Common.MoveObjects>(),
    Parameters = new Dictionary<string, object>(),
    SubReportDataTables = new List<Majorsilence.CrystalCmd.Common.SubReports>()
};

// add as many data tables as needed. The client library will do the necessary conversions to json/cs
reportData.AddData("report name goes here", "table name goes here", dt);

// export to pdf
var crystalReport = System.IO.File.ReadAllBytes("The rpt template file path goes here");
using (var instream = new MemoryStream(crystalReport))
using (var outstream = new MemoryStream())
{
    //var rpt = new Majorsilence.CrystalCmd.Client.Report(serverUrl, username: "The server username goes here");
    var rpt = new Majorsilence.CrystalCmd.Client.ReportWithPolling(serverUrl, username: "The server username goes here");
    using (var stream = await rpt.GenerateAsync(reportData, instream, _httpClient))
    {
        stream.CopyTo(outstream);
        return outstream.ToArray();
    }
}
```

Part III

Data

Chapter 6

SQLite

SQLite is a database that is a file. There is no server to install, no port to open, no user to create; an application opens the file and starts querying. That single property makes it the right choice far more often than people expect: desktop application storage, a cache, test fixtures, and small web applications all work well on it.

This chapter shows the same small task three ways - raw ADO.NET, Dapper, and Entity Framework Core - because those three layers reappear for every other database in this part of the book. Seeing them against the simplest possible store makes the differences between them easy to see. [Data Access in .NET](#) covers each in more depth.

6.1 SQLite

SQLite is a lightweight, serverless, self-contained SQL database engine. It stores the entire database as a single file on disk, requires no separate server process, and is included in .NET by default. SQLite is ideal for development, prototyping, desktop, mobile, and small-to-medium web applications.

Why use SQLite for new projects?

- **Zero configuration:** No server setup or management required.
- **Easy to use:** Simple file-based deployment—just copy the database file.
- **Reliable and fast:** ACID-compliant and performant for most workloads.
- **Portable:** Works across platforms (Windows, Linux, macOS).
- **Scalable for prototyping:** Start with SQLite, then migrate to a larger DBMS, PostgreSQL, if/when needed.

See [Why you should probably be using SQLite](#).

6.2 C# Examples

Install the NuGet package:

```
dotnet add package Microsoft.Data.Sqlite
```

Create and query a database:

```
using Microsoft.Data.Sqlite;

var connectionString = "Data Source=tvshows.db";
using var connection = new SqliteConnection(connectionString);
connection.Open();
```

```
// Create table
var createCmd = connection.CreateCommand();
createCmd.CommandText = @"
    CREATE TABLE IF NOT EXISTS TvShows (
        Id INTEGER PRIMARY KEY AUTOINCREMENT,
        ShowName TEXT NOT NULL,
        Rating REAL
    );";
createCmd.ExecuteNonQuery();

// Insert data
var insertCmd = connection.CreateCommand();
insertCmd.CommandText = "INSERT INTO TvShows (ShowName, Rating) VALUES ($name, $rating);";
insertCmd.Parameters.AddWithValue("$name", "Friends");
insertCmd.Parameters.AddWithValue("$rating", 4.8);
insertCmd.ExecuteNonQuery();

// Query data
var selectCmd = connection.CreateCommand();
selectCmd.CommandText = "SELECT Id, ShowName, Rating FROM TvShows;";
using var reader = selectCmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader.GetInt32(0)}: {reader.GetString(1)} ({reader.GetDouble(2)})");
}
```

Note: For more advanced scenarios, consider using [Dapper](#) or Entity Framework Core with SQLite as the provider.

6.3 SQLite with Dapper

Install NuGet packages:

```
dotnet add package Dapper
dotnet add package Microsoft.Data.Sqlite
```

Example: Querying SQLite with Dapper

```
using System;
using System.Collections.Generic;
using Dapper;
using Microsoft.Data.Sqlite;

public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public double Rating { get; set; }
}

public class Example
{
    public IEnumerable<TvShow> GetShows()
    {
```

```

        using var conn = new SqlConnection("Data Source=tvshows.db");
        conn.Open();
        return conn.Query<TvShow>("SELECT Id, ShowName, Rating FROM TvShows WHERE Rating > @minRating");
    }
}

```

6.4 SQLite with Entity Framework Core

Install NuGet packages:

```

dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Sqlite

```

Example: DbContext and Model

```

using Microsoft.EntityFrameworkCore;

public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public double Rating { get; set; }
}

public class AppDbContext : DbContext
{
    public DbSet<TvShow> TvShows { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlite("Data Source=tvshows.db");
}

// Usage
using var db = new AppDbContext();
db.TvShows.Add(new TvShow { ShowName = "Friends", Rating = 4.8 });
db.SaveChanges();

var highRated = db.TvShows.Where(t => t.Rating > 4.0).ToList();

```


Chapter 7

PostgreSQL

PostgreSQL is the default choice for a serious relational database when nothing forces another one. It is open source, runs everywhere, and has spent decades accumulating features - JSON columns, full text search, window functions, extensions like PostGIS - without giving up correctness.

The structure here mirrors **SQLite**: install it, then the same work through raw ADO.NET with Npgsql, through Dapper, and through Entity Framework Core. The .NET side barely changes between the two databases, which is the point of the provider model.

7.1 PostgreSQL - Install

Follow the instructions found at <https://www.postgresql.org/download/>.

See the majorsilence **PostgreSQL** page for fedora and ubuntu configuration instructions instructions.

For managing PostgreSQL databases use **pgAdmin**.

7.2 PostgreSQL Examples

7.2.1 Create a Table

```
CREATE TABLE tv_shows (  
    id SERIAL PRIMARY KEY,  
    show_name VARCHAR(100) NOT NULL,  
    rating NUMERIC(3,1)  
);
```

7.2.2 Insert Data

```
INSERT INTO tv_shows (show_name, rating) VALUES ('Friends', 4.8);  
INSERT INTO tv_shows (show_name, rating) VALUES ('Dexter', 4.5);
```

7.2.3 Stored Procedure

A stored procedure to insert a new TV show:

```
CREATE OR REPLACE PROCEDURE insert_tv_show(p_show_name VARCHAR, p_rating NUMERIC)  
LANGUAGE plpgsql  
AS $$
```

```
BEGIN
    INSERT INTO tv_shows (show_name, rating) VALUES (p_show_name, p_rating);
END;
$$;
```

Call the procedure:

```
CALL insert_tv_show('Frasier', 4.6);
```

7.2.4 Stored Function

A function to get the average rating:

```
CREATE OR REPLACE FUNCTION get_average_rating()
RETURNS NUMERIC AS $$
BEGIN
    RETURN (SELECT AVG(rating) FROM tv_shows);
END;
$$ LANGUAGE plpgsql;
```

Usage:

```
SELECT get_average_rating();
```

7.2.5 View

A view showing only highly rated shows:

```
CREATE OR REPLACE VIEW high_rated_shows AS
SELECT id, show_name, rating
FROM tv_shows
WHERE rating >= 4.5;
```

Query the view:

```
SELECT * FROM high_rated_shows;
```

7.2.6 C# Example: Querying PostgreSQL

Install the [Npgsql](#) NuGet package:

```
dotnet add package Npgsql
```

Sample C# code:

```
using Npgsql;

var connString = "Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb";
using var conn = new NpgsqlConnection(connString);
conn.Open();

// Query data
using var cmd = new NpgsqlCommand("SELECT id, show_name, rating FROM tv_shows", conn);
using var reader = cmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader.GetInt32(0)}: {reader.GetString(1)} ({reader.GetDecimal(2)})");
}
```

```
// Call a function
using var avgCmd = new NpgsqlCommand("SELECT get_average_rating()", conn);
var avg = avgCmd.ExecuteScalar();
Console.WriteLine($"Average rating: {avg}");
```

Note: For async usage, use `await conn.OpenAsync()` and `await cmd.ExecuteReaderAsync()`.

7.3 PostgreSQL with Dapper

Dapper is a lightweight ORM for .NET that works well with PostgreSQL via the **Npgsql** driver.

Install NuGet packages:

```
dotnet add package Dapper
dotnet add package Npgsql
```

Example: Querying PostgreSQL with Dapper

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Dapper;
using Npgsql;

public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public decimal Rating { get; set; }
}

public class Example
{
    public async Task<IEnumerable<TvShow>> GetShowsAsync()
    {
        var connString = "Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb";
        using var conn = new NpgsqlConnection(connString);
        await conn.OpenAsync();

        var sql = "SELECT id, show_name AS ShowName, rating FROM tv_shows WHERE rating > @minRating";
        return await conn.QueryAsync<TvShow>(sql, new { minRating = 4.0m });
    }
}
```

7.4 PostgreSQL with Entity Framework Core

Install NuGet packages:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Npgsql.EntityFrameworkCore.PostgreSQL
```

Example: DbContext and Model

```
using Microsoft.EntityFrameworkCore;
```

```
public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public decimal Rating { get; set; }
}

public class AppDbContext : DbContext
{
    public DbSet<TvShow> TvShows { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseNpgsql("Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb");
}

// Usage
using var db = new AppDbContext();
db.TvShows.Add(new TvShow { ShowName = "Friends", Rating = 4.8m });
db.SaveChanges();

var highRated = db.TvShows.Where(t => t.Rating > 4.0m).ToList();
```

Note:

- Use migrations to create/update your PostgreSQL schema:
dotnet ef migrations add InitialCreate
dotnet ef database update - See [Npgsql EF Core docs](#) for advanced usage.

Chapter 8

Microsoft SQL Server

All sql scripts included in this section expect to be run in your preferred sql tool. If you need to install sql server skip to [SQL - Install](#).

Azure Data Studio, which earlier versions of this text recommended, was retired by Microsoft and stopped receiving support in February 2026. See [Client tools](#) for what to use instead.

8.1 Adventure Works

While many of the sql examples shown will not use the adventure works sample database I suggest that it is restored and used to investigate sql server.

For a more detailed sample database download and restore [Microsoft's AdventureWorks database](#).

Before restoring the bak change the owner to mssql and move it to a folder that sql server has permissions to access.

```
sudo mkdir -p /var/opt/mssql/backup/  
sudo chown mssql /var/opt/mssql/backup/  
sudo chgrp mssql /var/opt/mssql/backup/  
chown mssql AdventureWorksLT2019.bak  
chgrp mssql AdventureWorksLT2019.bak  
sudo mv AdventureWorksLT2019.bak /var/opt/mssql/backup/
```

Find the logical names

```
USE [master];  
GO  
RESTORE FILELISTONLY  
FROM DISK = '/var/opt/mssql/backup/AdventureWorksLT2019.bak'
```

Restore the database.

```
USE [master];  
GO  
RESTORE DATABASE [AdventureWorks2019]  
FROM DISK = '/var/opt/mssql/backup/AdventureWorksLT2019.bak'  
WITH  
    MOVE 'AdventureWorksLT2012_Data' TO '/var/opt/mssql/data/AdventureWorks2019.mdf',  
    MOVE 'AdventureWorksLT2012_Log' TO '/var/opt/mssql/data/AdventureWorks2019_log.ldf',  
    FILE = 1,  
    NOUNLOAD,
```

```
STATS = 5;  
GO
```

8.2 Create a database

```
use master;  
create database SqlPlayground;
```

8.3 Create a table

Create a table using a UNIQUEIDENTIFIER (sequential guid) column as the primary key.

```
use SqlPlayground;  
  
create table [dbo].[TvShows]  
(  
    Id UNIQUEIDENTIFIER NOT NULL PRIMARY KEY DEFAULT NEWSEQUENTIALID(),  
    ShowName nvarchar(50) not null,  
    ShowLength int not null,  
    Summary nvarchar(max) not null,  
    Rating decimal(18,2) null,  
    Episode nvarchar(200) not null,  
    ParentalGuide nvarchar(5) null  
)
```

As an alternative, create the table with a bigint identity column as the primary key.

```
create table [dbo].[TvShows]  
(  
    Id BIGINT NOT NULL IDENTITY PRIMARY KEY,  
    ShowName nvarchar(50) not null,  
    ShowLength int not null,  
    Summary nvarchar(max) not null,  
    Rating decimal(18,2) null,  
    Episode nvarchar(200) not null,  
    ParentalGuide nvarchar(5) null  
)
```

Note: schemas, tables, and column names can be surrounded in square brackets []. This is for when special characters or reserved keywords are part of the name.

8.4 Alter a table

```
alter table TvShows  
add FirstAiredUtc DateTime;
```

8.5 Create indexes

Review [Clustered and nonclustered indexes described](#) and [CREATE INDEX](#).

```
create index index_tvshows_showname ON dbo.TvShows (ShowName);
```

8.6 SELECT

```
select * from TvShows;
select * from TvShows where ShowName = 'Dexter';
select Id, ShowName, ShowLength, Summary, FirstAiredUtc
from TvShows;
```

8.7 INSERT

Insert a new row into a table.

```
insert into TvShows (ShowName, ShowLength, Summary, Rating, Episode, ParentalGuide)
values ('Frasier', '30', 'Frasier goes home.', 4.56, '1e01', 'PG');
```

Insert a new row into a table and select back the new unique identifier of that row.

```
declare @InsertedRowIds table(InsertedId UNIQUEIDENTIFIER);

insert into TvShows (ShowName, ShowLength, Summary, Rating, Episode, ParentalGuide)
OUTPUT inserted.Id INTO @InsertedRowIds(InsertedId)
values ('Frasier', '30', 'Frasier does it again.', 3.68, '2e01', 'PG');

select * FROM @InsertedRowIds;
```

Insert a new row into a table that uses a bigint identity column and select back the new id of that row.

```
insert into TvShows (ShowName, ShowLength, Summary, Rating, Episode, ParentalGuide)
values ('Frasier', '30', 'Frasier does it again.', 3.68, '2e01', 'PG');

select SCOPE_IDENTITY();
```

Further reading

- [INSERT](#)
- [OUTPUT clause](#)

8.8 UPDATE

When executing updates be sure to include a where clause to avoid updating every record in a table.

```
update TvShows set ParentalGuide = 'PG13' where ShowName='Friends';
update TvShows set ParentalGuide = 'PG' where ShowName = 'Frasier';
update TvShows set ParentalGuide = '18A' where ShowName = 'Dexter';
update TvShows set ParentalGuide = 'PG' where ShowName in ('Friends', 'Frasier');
```

8.9 DELETE

When executing deletes be sure to include a where clause to avoid deleting every record in a table.

```
delete from TvShows where ShowName = 'Dexter';
```

8.10 Foreign Keys

A **foreign key** is a constraint that enforces a relationship between columns in two tables, ensuring that the value in one table matches a value in another. This maintains referential integrity between related

data.

For example, suppose you have a `TvShows` table and an `Episodes` table. Each episode references a TV show by its `TvShowId`:

```
CREATE TABLE TvShows (
    Id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    ShowName NVARCHAR(50) NOT NULL
);

CREATE TABLE Episodes (
    Id BIGINT NOT NULL IDENTITY PRIMARY KEY,
    TvShowId BIGINT NOT NULL,
    EpisodeName NVARCHAR(100) NOT NULL,
    FOREIGN KEY (TvShowId) REFERENCES TvShows(Id)
);
```

In this example, `Episodes.TvShowId` must match an existing `TvShows.Id`, ensuring episodes are always linked to a valid TV show.

8.11 JOIN

A **JOIN** in SQL combines rows from two or more tables based on a related column between them. The most common type is an **INNER JOIN**, which returns only the rows where there is a match in both tables.

Example:

Suppose you have `TvShows` and `Episodes` tables. To list all episodes with their show names:

```
SELECT
    TvShows.ShowName,
    Episodes.EpisodeName
FROM
    TvShows
INNER JOIN
    Episodes ON TvShows.Id = Episodes.TvShowId;
```

This query returns each episode along with the name of its TV show.

8.12 CTE

A **Common Table Expression (CTE)** is a temporary result set in SQL that you can reference within a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement. CTEs make complex queries easier to read and maintain, and are especially useful for recursive queries or breaking down large queries into logical building blocks.

Example:

Suppose you want to select all TV shows with a rating above 4.0 and then count them.

```
WITH HighRatedShows AS (
    SELECT *
    FROM TvShows
    WHERE Rating > 4.0
)
SELECT COUNT(*) AS HighRatedShowCount
FROM HighRatedShows;
```

In this example, the CTE `HighRatedShows` selects all shows with a rating above 4.0, and the main query counts how many such shows exist.

8.13 Stored Procedures

A **stored procedure** in SQL Server is a precompiled collection of one or more T-SQL statements that can be executed as a single unit. Stored procedures help encapsulate logic, improve performance, and promote code reuse.

Example:

This stored procedure selects all TV shows with a rating above a specified value:

```
CREATE PROCEDURE GetHighRatedTvShows
    @MinRating DECIMAL(18,2)
AS
BEGIN
    SELECT *
    FROM TvShows
    WHERE Rating >= @MinRating;
END
```

To execute the procedure:

```
EXEC GetHighRatedTvShows @MinRating = 4.5;
```

This will return all rows from `TvShows` where the `Rating` is 4.5 or higher.

8.14 Stored Functions

A **stored function** in SQL Server is a user-defined function (UDF) that returns a single value or a table. Functions can be used in queries, computed columns, or as part of expressions. Unlike stored procedures, functions must return a value and cannot modify database state (no `INSERT`, `UPDATE`, or `DELETE`).

Example:

This scalar-valued function returns the full name of a TV show episode by combining the show name and episode name.

```
CREATE FUNCTION dbo.GetFullEpisodeName
(
    @ShowName NVARCHAR(50),
    @EpisodeName NVARCHAR(100)
)
RETURNS NVARCHAR(200)
AS
BEGIN
    RETURN @ShowName + ' - ' + @EpisodeName
END
```

Usage:

```
SELECT dbo.GetFullEpisodeName('Friends', 'The One Where It All Began') AS FullEpisodeName;
```

8.15 Views

A **view** in SQL Server is a virtual table based on the result of a `SELECT` query. Views simplify complex queries, encapsulate logic, and can help restrict access to specific data.

Example:

Create a view that lists only TV shows with a rating above 4.0:

```
CREATE VIEW HighRatedTvShows AS
SELECT Id, ShowName, Rating
FROM TvShows
WHERE Rating > 4.0;
```

You can then query the view like a table:

```
SELECT * FROM HighRatedTvShows;
```

8.16 SQL - Install

8.16.1 SQL server windows install

Download [sql server](#) from Microsoft. The simple install method is to double click the setup.exe and use the user interface to complete the install.

If it is a non production environment, for development choose the developer edition.

If you wish to automate the install it can be script with options similar to the below example.

```
setup.exe /ACTION=INSTALL /IACCEPTSQLSERVERLICENSETERMS /FEATURES="SQL,Tools" /SECURITYMODE=SQL /SAPWD
```

Review the [Install SQL Server on Windows from the command prompt](#) page for up to date options and documentation.

8.16.2 SQL server linux install

See [Quickstart: Install SQL Server and create a database on Ubuntu](#) for further details.

Run these commands to install sql server 2022 on a ubuntu server.

```
# Download the Microsoft repository GPG keys
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo tee /etc/apt/trusted.gpg.d/microsoft.asc
sudo add-apt-repository "$ (wget -qO- https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/prod.list)"
# Update the list of packages after we added packages.microsoft.com
sudo apt-get update
# Install SQL Server
sudo apt-get install mssql-server
sudo /opt/mssql/bin/mssql-conf setup
```

To enable the sql agent feature run this command:

```
sudo /opt/mssql/bin/mssql-conf set sqlagent.enabled true
sudo systemctl restart mssql-server
```

If the command line tools are also required run these commands:

```
wget -qO- https://packages.microsoft.com/keys/microsoft.asc | sudo tee /etc/apt/trusted.gpg.d/microsoft.asc
wget -qO- https://packages.microsoft.com/config/ubuntu/$(lsb_release -rs)/prod.list | sudo tee /etc/apt/sources.list.d/mssql-tools.list
sudo apt-get update
sudo apt-get install mssql-tools unixodbc-dev

echo 'export PATH="$PATH:/opt/mssql-tools/bin"' >> ~/.bash_profile
```

8.16.3 SQL server extra configuration after the install

Set some initial configuration options from any of the tools in [Client tools](#). Run the following sql.

```
sp_configure 'show advanced options', 1
reconfigure with override

sp_configure 'max server memory (MB)', -- 90% of OS MEM
reconfigure with override
```

The per database settings worth changing are all reachable from T-SQL, which means the same script works from every tool and on Linux, where there is no Configuration Manager to click through.

```
-- Full recovery keeps point in time restore available. If the data is non
-- production or not important, SIMPLE is fine and the log stops growing.
ALTER DATABASE YourDatabase SET RECOVERY FULL;

-- Percentage growth on both files. The default 1 MB log growth produces
-- thousands of tiny virtual log files on a busy database.
ALTER DATABASE YourDatabase MODIFY FILE (NAME = 'YourDatabase', FILEGROWTH = 10%);
ALTER DATABASE YourDatabase MODIFY FILE (NAME = 'YourDatabase_log', FILEGROWTH = 10%);

-- Match the engine version so the current query optimiser is used.
ALTER DATABASE YourDatabase SET COMPATIBILITY_LEVEL = 170; -- SQL Server 2025

-- Query Store, read write, so it is actually collecting.
ALTER DATABASE YourDatabase SET QUERY_STORE = ON;
ALTER DATABASE YourDatabase SET QUERY_STORE (OPERATION_MODE = READ_WRITE);
```

On Windows, force encryption lives in SQL Server Configuration Manager under Protocols. On Linux, set it with mssql-conf instead.

```
sudo /opt/mssql/bin/mssql-conf set network.forceencryption 1
sudo systemctl restart mssql-server
```

8.17 Reference - Admin

Use [spBlitz \(SQL First Responder Kit\)](#) to detect problems with sql server. Follow the instructions on the spBlitz site.

A few examples:

```
-- Realtime performance advice that should be run first when responding to an incident or case
exec sp_BlitzFirst

-- Overall Health Check
exec sp_Blitz

-- Find the Most Resource-Intensive Queries
exec sp_BlitzCache

-- Tune Your Indexes
exec sp_BlitzIndex
```

Use the [Ola Hallengren SQL Server Maintenance Solutions](#) for excellent pre-made community backed maintenance jobs.

Both of those install as stored procedures, so they are run the same way from any client, and scheduled with SQL Server Agent jobs.

8.18 Client tools

Azure Data Studio was the cross platform answer here for years. Microsoft retired it, with support ending in February 2026, and folded its functionality into the VS Code extension below. Anything still recommending it is out of date.

- **MSSQL extension for VS Code** - Microsoft's replacement for Azure Data Studio, and where the effort now goes. Free, runs on linux, mac and windows, and includes query editing, a results grid, schema browsing and query plan viewing.
- **Beekeeper Studio** - open source, cross platform, and the most pleasant of these to actually sit in front of. Covers SQL Server, PostgreSQL, SQLite and Redis, so like DBeaver it is one tool for most of the **Data** part of this book. The free community edition is enough for querying and editing day to day; the paid edition adds things like the query magics and backup/restore UI.
- **DBeaver** - open source and cross platform, and the one to pick when breadth matters most. It reaches databases nothing else here does, and the community edition is fully featured, at the cost of a heavier, more cluttered interface than Beekeeper's.
- **DataGrip** - JetBrains, commercial, cross platform, and included with an All Products subscription if you already use Rider.
- **SSMS** - still the most complete tool for administration, and still windows only. Reach for it when you need the deeper admin surface; do not build a workflow around it if your team is on linux or mac.
- **sqlcmd** - the command line client, and the one that works over ssh, inside a container, and in a build pipeline.

The lesson worth taking from the retirement is to prefer T-SQL over a GUI workflow when writing anything down. A script in source control outlives whichever client is fashionable.

8.19 SQL Profiler and Extended Events

The old SQL Profiler and its trace API are deprecated, and the graphical XEvent Profiler was an Azure Data Studio feature that went away with it. **Extended Events** is the supported mechanism, and because a session is created and read with T-SQL, it works from any client on any platform.

Capture statements that took longer than a second.

```
CREATE EVENT SESSION slow_queries ON SERVER
ADD EVENT sqlserver.sql_batch_completed (
    ACTION (sqlserver.client_app_name, sqlserver.database_name, sqlserver.sql_text)
    -- duration is in microseconds
    WHERE duration > 1000000
),
ADD EVENT sqlserver.rpc_completed (
    ACTION (sqlserver.client_app_name, sqlserver.database_name, sqlserver.sql_text)
    WHERE duration > 1000000
)
ADD TARGET package0.ring_buffer
WITH (MAX_MEMORY = 4096 KB, TRACK_CAUSALITY = ON, STARTUP_STATE = OFF);

ALTER EVENT SESSION slow_queries ON SERVER STATE = START;
```

The ring buffer target holds the results in memory. Read it by shredding the XML.

```

SELECT
    x.e.value('@timestamp')[1], 'datetime2') AS occurred,
    x.e.value('(data[@name="duration"]/value)[1]', 'bigint') / 1000 AS duration_ms,
    x.e.value('(action[@name="database_name"]/value)[1]', 'nvarchar(128)') AS database_name,
    x.e.value('(action[@name="client_app_name"]/value)[1]', 'nvarchar(256)') AS client_app,
    x.e.value('(action[@name="sql_text"]/value)[1]', 'nvarchar(max)') AS sql_text
FROM (
    SELECT CAST(st.target_data AS xml) AS target_data
    FROM sys.dm_xe_session_targets AS st
    JOIN sys.dm_xe_sessions AS s ON s.address = st.event_session_address
    WHERE s.name = 'slow_queries' AND st.target_name = 'ring_buffer'
) AS raw
CROSS APPLY raw.target_data.nodes('RingBufferTarget/event') AS x(e)
ORDER BY occurred DESC;

```

Stop it when finished. An event session left running on a busy server is overhead nobody remembers enabling.

```

ALTER EVENT SESSION slow_queries ON SERVER STATE = STOP;
DROP EVENT SESSION slow_queries ON SERVER;

```

Use a `package0.event_file` target instead of the ring buffer when a session should survive a restart or capture more than a few thousand events. For day to day “why was this slow”, the [Query Store](#) below is usually the better first stop, because it is always on and keeps its history.

8.20 SQL Query Store

Monitor performance by using the [Query Store](#)

8.21 SQL Watch

[SQL Watch](#) - sql monitor.

Chapter 9

Redis

Redis is an open-source, in-memory data store commonly used as a database, cache, and message broker. It supports data structures such as strings, hashes, lists, sets, and more, and is known for its high performance and simplicity.

forks:

- **Valkey**: A community-driven fork supported by the Linux Foundation, aiming to maintain an open-source alternative.
- **Redict**: Another fork focused on preserving the original open-source spirit of Redis.

These forks are compatible with Redis and are intended to provide drop-in replacements for users who require a fully open-source solution.

Other redis compatible solutions:

- **DragonflyDB**

9.1 Ubuntu Install

```
sudo apt update
sudo apt install redis-server
sudo cp /etc/redis/redis.conf /etc/redis/redis.conf.backup

sudo ufw allow ssh
sudo ufw allow redis
sudo ufw allow 6380/tcp
sudo ufw enable
sudo ufw status
```

9.1.1 Add redis password

/etc/redis/redis.conf

add

```
user default on >[PLACEHOLDER] ~* +@all
acl-pubsub-default allchannels
```

9.1.2 Connect to password protected redis server

```
redis-cli -h 127.0.0.1 -p 6379
AUTH [PLACEHOLDER]
```

9.1.3 expose to the network

```
sudo vim /etc/redis/redis.conf
bind 0.0.0.0
```

9.1.4 Restart on failure

Ensure restart on failure is enabled

```
sudo cat /lib/systemd/system/redis-server.service
```

2. **Look for Restart Policies:** Within the service file, look for directives related to the restart policy. Common directives include `Restart` and `RestartSec`.

Example:

```
[Service]
Type=notify
ExecStart=/usr/bin/redis-server /etc/redis/redis.conf
ExecStop=/usr/bin/redis-shutdown
User=redis
Group=redis
RuntimeDirectory=redis
RuntimeDirectoryMode=2755
PIDFile=/run/redis/redis-server.pid
TimeoutStopSec=0
Restart=on-failure
RestartSec=5
```

3. **Modify the Service Configuration if Necessary:** If the `Restart` directive is not set or you want to customize it, you can create a systemd override file to modify the service configuration without changing the original service file.

```
sudo systemctl edit redis-server
```

This command opens an editor where you can add or override directives. For example, to ensure the service restarts on failure, you can add:

```
[Unit]
StartLimitIntervalSec=0 # Disable the limit on the time window
StartLimitBurst=0       # Disable the limit on the number of restart attempts

[Service]
Restart=always          # Always restart the service
RestartSec=10           # Wait 10 seconds before restarting
```

4. **Reload Systemd and Restart the Service:** After making changes, reload the systemd configuration and restart the Redis service to apply the changes.

```
sudo systemctl daemon-reload
sudo systemctl restart redis-server
```

5. **Verify the Configuration:** Finally, you can verify that the service is configured correctly by checking its status.

```
sudo systemctl status redis-server
```

9.1.5 TLS Connection Support

If TLS support is enabled the client connection strings must be configured to use it.

9.1.5.1 1. Generate SSL/TLS Certificates

You can generate self-signed certificates for testing purposes or obtain certificates from a trusted Certificate Authority (CA) for production use. Here's how to generate self-signed certificates using OpenSSL:

```
# Create a directory to store the certificates
mkdir -p /etc/redis/ssl
cd /etc/redis/ssl

# Generate a private key
openssl genrsa -out redis-server.key 2048

# Generate a self-signed certificate
openssl req -new -x509 -key redis-server.key -out redis-server.crt -days 3652

# Generate a private key for the client
openssl genrsa -out redis-client.key 2048

# Generate a certificate signing request (CSR) for the client
openssl req -new -key redis-client.key -out redis-client.csr

# Generate a self-signed certificate for the client
openssl x509 -req -in redis-client.csr -CA redis-server.crt -CAkey redis-server.key -CAcreateserial

chgrp -R redis /etc/redis/ssl
chown -R redis /etc/redis/ssl
```

9.1.5.2 2. Configure Redis to Use TLS

Edit the Redis configuration file (`/etc/redis/redis.conf`) to enable TLS and specify the paths to your certificates and keys.

```
# Non-TLS port, disable for enhanced security
port 6379

# Enable TLS
tls-port 6380

# Specify the paths to the certificates and keys
tls-cert-file /etc/redis/ssl/redis-server.crt
tls-key-file /etc/redis/ssl/redis-server.key
tls-ca-cert-file /etc/redis/ssl/redis-server.crt

# Optional: Require clients to authenticate using a client certificate
tls-auth-clients no
```

9.1.5.3 3. Restart Redis Server

After making these changes, restart the Redis server to apply the new configuration.

```
sudo systemctl restart redis-server
```

9.2 Session

To use Redis for web sessions in C#, add the `Microsoft.Extensions.Caching.StackExchangeRedis` NuGet package to your ASP.NET Core project. In `Program.cs`, configure session state to use Redis as the backing store:

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost:6379,password=yourpassword";
    options.InstanceName = "SampleInstance";
});

builder.Services.AddSession(options =>
{
    options.Cookie.HttpOnly = true;
    options.Cookie.IsEssential = true;
    options.IdleTimeout = TimeSpan.FromMinutes(30);
});

app.UseSession();
```

You can then use `HttpContext.Session` to store and retrieve session data. Redis will persist session state across web server restarts and scale-out scenarios.

9.3 Cache

To use Redis as a distributed cache in a C# ASP.NET Core application, add the `Microsoft.Extensions.Caching.StackExchangeRedis` NuGet package. Configure Redis in `Program.cs`:

```
builder.Services.AddStackExchangeRedisCache(options =>
{
    options.Configuration = "localhost:6379,password=yourpassword";
    options.InstanceName = "SampleInstance";
});
```

You can then use `IDistributedCache` to store and retrieve cached data:

```
public class MyController : Controller
{
    private readonly IDistributedCache _cache;

    public MyController(IDistributedCache cache)
    {
        _cache = cache;
    }

    public async Task<IActionResult> Index()
    {
        var value = await _cache.GetStringAsync("myKey");
        if (value == null)
```

```

    {
        value = "Hello from Redis cache!";
        await _cache.SetStringAsync("myKey", value, new DistributedCacheEntryOptions
        {
            AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(10)
        });
    }
    return Content(value);
}
}

```

This enables fast, centralized caching for web applications, improving performance and scalability.

9.4 Publish and Subscribe

Publish/Subscribe (Pub/Sub) is a messaging pattern where senders (publishers) send messages to channels without knowing who will receive them, and receivers (subscribers) listen for messages on those channels. Redis provides built-in support for Pub/Sub, enabling real-time messaging between distributed components.

9.4.1 Using Redis Pub/Sub in C

To use Redis Pub/Sub in a C# application, add the `StackExchange.Redis` NuGet package. You can then publish and subscribe to messages as shown below:

```

using StackExchange.Redis;

var redis = ConnectionMultiplexer.Connect("localhost:6379,password=yourpassword");
var pubsub = redis.GetSubscriber();

// Subscribe to a channel
pubsub.Subscribe("notifications", (channel, message) => {
    Console.WriteLine($"Received: {message}");
});

// Publish a message to the channel
pubsub.Publish("notifications", "Hello from publisher!");

// Keep the application running to receive messages
Console.ReadLine();

```

This allows different parts of your application (or different applications) to communicate in real time using Redis channels.

9.5 Example: Redis Pub/Sub with Web Frontend and Worker Services

This example demonstrates a simple architecture where:

- A web frontend allows users to submit work (e.g., a “task”).
- The frontend publishes the task to a Redis channel.
- One or more worker services (in separate processes) subscribe to the channel, process the task, and publish a completion message to another channel.
- The frontend listens for completion messages and notifies the user when their task is done.

9.5.1 1. Web Frontend (ASP.NET Core + JavaScript)

Backend Controller (C#):

```
// Controller to accept user input and publish to Redis
[ApiController]
[Route("api/[controller]")]
public class TasksController : ControllerBase
{
    private readonly IConnectionMultiplexer _redis;
    public TasksController(IConnectionMultiplexer redis) => _redis = redis;

    [HttpPost]
    public async Task<IActionResult> SubmitTask([FromBody] TaskRequest request)
    {
        var id = Guid.NewGuid().ToString();
        var pub = _redis.GetSubscriber();
        await pub.PublishAsync("tasks", $"{id}:{request.Payload}");
        return Ok(new { taskId = id });
    }
}

public class TaskRequest
{
    public string Payload { get; set; }
}
```

Frontend (HTML + JavaScript):

```
<input id="taskInput" placeholder="Enter task" />
<button onclick="submitTask()">Submit</button>
<div id="status"></div>
<script>
let taskId = null;
function submitTask() {
    const payload = document.getElementById('taskInput').value;
    fetch('/api/tasks', {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body: JSON.stringify({ payload })
    })
    .then(r => r.json())
    .then(data => {
        taskId = data.taskId;
        document.getElementById('status').innerText = 'Task submitted. Waiting for completion...';
    });
}

// Listen for completion via WebSocket (SignalR or custom implementation)
const ws = new WebSocket('wss://yourserver/ws');
ws.onmessage = function(event) {
    const msg = JSON.parse(event.data);
    if (msg.taskId === taskId) {
        document.getElementById('status').innerText = 'Task complete: ' + msg.result;
    }
};
```

```
</script>
```

Note: The backend should push completion messages to the frontend via WebSocket (e.g., using SignalR).

9.5.2 2. Worker Service (C# Console App)

```
using StackExchange.Redis;

var redis = ConnectionMultiplexer.Connect("localhost:6379,password=yourpassword");
var sub = redis.GetSubscriber();

sub.Subscribe("tasks", async (channel, message) => {
    var parts = message.ToString().Split(':', 2);
    var taskId = parts[0];
    var payload = parts[1];

    // Simulate work
    await Task.Delay(2000);
    var result = payload.ToUpperInvariant();

    // Publish completion
    await sub.PublishAsync("tasks-complete", $"{taskId}:{result}");
});

Console.WriteLine("Worker running. Press Enter to exit.");
Console.ReadLine();
```

9.5.3 3. Completion Notification Service

A background service (e.g., in your ASP.NET Core app) subscribes to "tasks-complete" and pushes updates to the frontend via WebSocket/SignalR.

```
public class CompletionNotifier : BackgroundService
{
    private readonly IConnectionMultiplexer _redis;
    private readonly IHubContext<NotifyHub> _hub;
    public CompletionNotifier(IConnectionMultiplexer redis, IHubContext<NotifyHub> hub)
    {
        _redis = redis; _hub = hub;
    }

    protected override Task ExecuteAsync(CancellationToken stoppingToken)
    {
        var sub = _redis.GetSubscriber();
        return sub.SubscribeAsync("tasks-complete", async (ch, msg) => {
            var parts = msg.ToString().Split(':', 2);
            var taskId = parts[0];
            var result = parts[1];
            await _hub.Clients.All.SendAsync("TaskCompleted", new { taskId, result });
        });
    }
}
```

SignalR Hub:

```
public class NotifyHub : Hub { }
```

Frontend SignalR Listener:

```
const connection = new signalR.HubConnectionBuilder().withUrl("/notifyhub").build();
connection.on("TaskCompleted", function(msg) {
    if (msg.taskId === taskId) {
        document.getElementById('status').innerText = 'Task complete: ' + msg.result;
    }
});
connection.start();
```

This pattern enables real-time user feedback for background work using Redis Pub/Sub, web frontend, and worker services.

9.6 Work and Message Queue with Redis in C

A work queue (also known as a message queue or task queue) allows producers to enqueue work items, and one or more consumers (workers) to process them asynchronously. Redis is commonly used for this pattern using its list commands (LPUSH/RPUSH to enqueue, BRPOP/BLPOP to dequeue).

9.6.1 Producer Example (Enqueue Work)

```
using StackExchange.Redis;

var redis = ConnectionMultiplexer.Connect("localhost:6379,password=yourpassword");
var db = redis.GetDatabase();

// Enqueue a work item (e.g., a JSON string or simple string)
await db.ListRightPushAsync("work-queue", "do_something:12345");
```

9.6.2 Consumer Example (Worker)

```
using StackExchange.Redis;

var redis = ConnectionMultiplexer.Connect("localhost:6379,password=yourpassword");
var db = redis.GetDatabase();

while (true)
{
    // Atomically move an item off the work queue and onto a processing
    // queue, so the item is not lost if this worker crashes mid-job.
    var result = await db.ListRightPopLeftPushAsync("work-queue", "processing-queue");
    if (!result.HasValue)
    {
        // Nothing waiting. StackExchange.Redis has no blocking pop, so
        // back off briefly instead of spinning on an empty queue.
        await Task.Delay(TimeSpan.FromMilliseconds(500));
        continue;
    }

    var workItem = result.ToString();
    Console.WriteLine($"Processing: {workItem}");
}
```

```
// Do work here...  
  
// Remove from processing-queue only after successful processing  
await db.ListRemoveAsync("processing-queue", workItem);  
}
```

Notes: - Use `ListRightPushAsync` to enqueue work and `ListRightPopLeftPushAsync` to dequeue it. - `StackExchange.Redis` deliberately does not expose the blocking `BRPOP`/`BLPOP` commands, because a blocked connection would stall every other operation sharing that multiplexer. Poll with a short delay, as above. - The “processing” queue is what makes this reliable. Anything left sitting in it belongs to a worker that died and can be moved back to `work-queue` by a reaper process. - For more advanced scenarios, consider libraries like [Hangfire](#).

Chapter 10

Data Access in .NET

Every database chapter in this part showed the same three approaches. This one takes them apart properly.

At the bottom is ADO.NET - `DbConnection`, `DbCommand`, `DbTransaction` - which every provider implements and every higher layer is built on. In the middle is Dapper, which maps query results onto objects and leaves the SQL to you. At the top is Entity Framework Core, which generates the SQL as well and adds change tracking and migrations.

None of them is the correct answer. Knowing all three is what lets you drop a layer when a query needs hand written SQL, without abandoning the ORM everywhere else.

10.1 DbConnection

A `DbConnection` represents an open connection to a database. It is the base class for database-specific connection classes like `SqlConnection` (SQL Server), `NpgsqlConnection` (PostgreSQL), and `SQLiteConnection` (SQLite).

Example: Using DbConnection with SQL Server

```
using System.Data.Common;
using Microsoft.Data.SqlClient;

string connectionString = "Server=localhost;Database=SqlPlayground;User Id=sa;Password=yourpassword";
using DbConnection conn = new SqlConnection(connectionString);
conn.Open();

using var cmd = conn.CreateCommand();
cmd.CommandText = "SELECT COUNT(*) FROM TvShows";
var count = cmd.ExecuteScalar();
Console.WriteLine($"Number of TV shows: {count}");
```

Example: Using DbConnection with PostgreSQL

```
using System.Data.Common;
using Npgsql;

string connectionString = "Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb";
using DbConnection conn = new NpgsqlConnection(connectionString);
conn.Open();
```

```
using var cmd = conn.CreateCommand();
cmd.CommandText = "SELECT COUNT(*) FROM tv_shows";
var count = cmd.ExecuteScalar();
Console.WriteLine($"Number of TV shows: {count}");
```

Example: Using DbConnection with SQLite

```
using System.Data.Common;
using Microsoft.Data.Sqlite;

string connectionString = "Data Source=tvshows.db";
using DbConnection conn = new SqliteConnection(connectionString);
conn.Open();

using var cmd = conn.CreateCommand();
cmd.CommandText = "SELECT COUNT(*) FROM TvShows";
var count = cmd.ExecuteScalar();
Console.WriteLine($"Number of TV shows: {count}");
```

Note: Always dispose connections (use `using` or `await using` for `async`) to free resources.

10.2 DbCommand

A `DbCommand` represents a SQL statement or stored procedure to execute against a database. It is the base class for provider-specific commands like `SqlCommand` (SQL Server), `NpgsqlCommand` (PostgreSQL), and `SQLiteCommand` (SQLite).

Example: Using DbCommand with SQL Server

```
using System.Data.Common;
using Microsoft.Data.SqlClient;

string connectionString = "Server=localhost;Database=SqlPlayground;User Id=sa;Password=yourpassword;";
using DbConnection conn = new SqlConnection(connectionString);
conn.Open();

using DbCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT ShowName, Rating FROM TvShows WHERE Rating > @minRating";
var param = cmd.CreateParameter();
param.ParameterName = "@minRating";
param.Value = 4.0m;
cmd.Parameters.Add(param);

using var reader = cmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader.GetString(0)} ({reader.GetDecimal(1)})");
}
```

Example: Using DbCommand with PostgreSQL

```
using System.Data.Common;
using Npgsql;

string connectionString = "Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb";
using DbConnection conn = new NpgsqlConnection(connectionString);
```

```

conn.Open();

using DbCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT show_name, rating FROM tv_shows WHERE rating > @minRating";
var param = cmd.CreateParameter();
param.ParameterName = "@minRating";
param.Value = 4.0m;
cmd.Parameters.Add(param);

using var reader = cmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader.GetString(0)} ({reader.GetDecimal(1)})");
}

```

Example: Using DbCommand with SQLite

```

using System.Data.Common;
using Microsoft.Data.Sqlite;

string connectionString = "Data Source=tvshows.db";
using DbConnection conn = new SqliteConnection(connectionString);
conn.Open();

using DbCommand cmd = conn.CreateCommand();
cmd.CommandText = "SELECT ShowName, Rating FROM TvShows WHERE Rating > $minRating";
var param = cmd.CreateParameter();
param.ParameterName = "$minRating";
param.Value = 4.0;
cmd.Parameters.Add(param);

using var reader = cmd.ExecuteReader();
while (reader.Read())
{
    Console.WriteLine($"{reader.GetString(0)} ({reader.GetDouble(1)})");
}

```

Note:

- Always use parameters to avoid SQL injection. - Use `ExecuteReader()` for queries, `ExecuteNonQuery()` for inserts/updates/deletes, and `ExecuteScalar()` for single-value results. - Dispose commands and readers properly (using `using` statements).

10.3 DataAdapters

A `DataAdapter` acts as a bridge between a `DataSet` and a database, allowing you to fill in-memory tables and update the database with changes. It is commonly used in ADO.NET for disconnected data access.

Example: Using SqlDataAdapter with SQL Server

```

using System.Data;
using Microsoft.Data.SqlClient;

string connectionString = "Server=localhost;Database=SqlPlayground;User Id=sa;Password=yourpassword";
using var conn = new SqlConnection(connectionString);
using var adapter = new SqlDataAdapter("SELECT * FROM TvShows", conn);

```

```

var dataSet = new DataSet();
adapter.Fill(dataSet, "TvShows");

// Access data
foreach (DataRow row in dataSet.Tables["TvShows"].Rows)
{
    Console.WriteLine($"{row["ShowName"]} ({row["Rating"]})");
}

```

Example: Updating Data with SqlDataAdapter

```

using System.Data;
using Microsoft.Data.SqlClient;

string connectionString = "Server=localhost;Database=SqlPlayground;User Id=sa;Password=yourpassword;";
using var conn = new SqlConnection(connectionString);
using var adapter = new SqlDataAdapter("SELECT * FROM TvShows", conn);

// Auto-generate commands for update/insert/delete
var builder = new SqlCommandBuilder(adapter);

var dataSet = new DataSet();
adapter.Fill(dataSet, "TvShows");

// Modify data in-memory
var table = dataSet.Tables["TvShows"];
table.Rows[0]["Rating"] = 5.0m;

// Push changes back to the database
adapter.Update(dataSet, "TvShows");

```

Example: Using SQLiteDataAdapter with SQLite

Microsoft.Data.Sqlite does not ship a DataAdapter. The older System.Data.SQLite provider does.

```

using System.Data;
using System.Data.SQLite;

string connectionString = "Data Source=tvshows.db";
using var conn = new SQLiteConnection(connectionString);
using var adapter = new SQLiteDataAdapter("SELECT * FROM TvShows", conn);

var dataSet = new DataSet();
adapter.Fill(dataSet, "TvShows");

```

If you are already using Microsoft.Data.Sqlite and only need a DataTable, load one from the reader instead of pulling in a second provider.

```

using System.Data;
using Microsoft.Data.Sqlite;

using var conn = new SqliteConnection("Data Source=tvshows.db");
conn.Open();

using var cmd = conn.CreateCommand();
cmd.CommandText = "SELECT * FROM TvShows";

```

```
var table = new DataTable();
using var reader = cmd.ExecuteReader();
table.Load(reader);
```

Notes: - DataAdapters are best for simple, disconnected scenarios. - For large-scale or modern applications, consider using ORMs like Dapper or Entity Framework. - Always dispose connections and adapters properly.

10.4 DbTransaction

A DbTransaction represents a database transaction, allowing you to execute multiple operations as a single unit of work. If any operation fails, you can roll back all changes to maintain data integrity.

Example: Using DbTransaction with SQL Server

```
using System.Data.Common;
using Microsoft.Data.SqlClient;

string connectionString = "Server=localhost;Database=SqlPlayground;User Id=sa;Password=yourpassword";
using DbConnection conn = new SqlConnection(connectionString);
conn.Open();

using var transaction = conn.BeginTransaction();
try
{
    using var cmd1 = conn.CreateCommand();
    cmd1.Transaction = transaction;
    cmd1.CommandText = "INSERT INTO TvShows (ShowName, ShowLength, Summary, Rating, Episode, Parent";
    cmd1.Parameters.Add(new SqlParameter("@name", "New Show"));
    cmd1.Parameters.Add(new SqlParameter("@length", 1200));
    cmd1.Parameters.Add(new SqlParameter("@summary", "A new show summary"));
    cmd1.Parameters.Add(new SqlParameter("@rating", 4.5m));
    cmd1.Parameters.Add(new SqlParameter("@episode", "1x01"));
    cmd1.Parameters.Add(new SqlParameter("@guide", "PG"));
    cmd1.ExecuteNonQuery();

    using var cmd2 = conn.CreateCommand();
    cmd2.Transaction = transaction;
    cmd2.CommandText = "UPDATE TvShows SET Rating = Rating + 0.1 WHERE ShowName = @name";
    cmd2.Parameters.Add(new SqlParameter("@name", "New Show"));
    cmd2.ExecuteNonQuery();

    transaction.Commit();
    Console.WriteLine("Transaction committed.");
}
catch
{
    transaction.Rollback();
    Console.WriteLine("Transaction rolled back.");
}
```

Example: Using DbTransaction with PostgreSQL

```
using System.Data.Common;
using Npgsql;
```

```

string connectionString = "Host=localhost;Username=postgres;Password=yourpassword;Database=yourdb";
using DbConnection conn = new NpgsqlConnection(connectionString);
conn.Open();

using var transaction = conn.BeginTransaction();
try
{
    using var cmd = conn.CreateCommand();
    cmd.Transaction = transaction;
    cmd.CommandText = "INSERT INTO tv_shows (show_name, rating) VALUES (@name, @rating)";
    cmd.Parameters.Add(new NpgsqlParameter("@name", "Another Show"));
    cmd.Parameters.Add(new NpgsqlParameter("@rating", 4.2m));
    cmd.ExecuteNonQuery();

    transaction.Commit();
}
catch
{
    transaction.Rollback();
}

```

Example: Using DbTransaction with SQLite

```

using System.Data.Common;
using Microsoft.Data.Sqlite;

string connectionString = "Data Source=tvshows.db";
using DbConnection conn = new SqliteConnection(connectionString);
conn.Open();

using var transaction = conn.BeginTransaction();
try
{
    using var cmd = conn.CreateCommand();
    cmd.Transaction = transaction;
    cmd.CommandText = "UPDATE TvShows SET Rating = Rating + 0.1 WHERE ShowName = $name";
    // conn is declared as DbConnection, so cmd.Parameters is a
    // DbParameterCollection, which has no AddWithValue. CreateParameter is the
    // provider neutral equivalent. Declare conn as SqliteConnection instead and
    // AddWithValue is available again.
    var nameParam = cmd.CreateParameter();
    nameParam.ParameterName = "$name";
    nameParam.Value = "Friends";
    cmd.Parameters.Add(nameParam);
    cmd.ExecuteNonQuery();

    transaction.Commit();
}
catch
{
    transaction.Rollback();
}

```

Notes: - Always associate commands with the transaction (`cmd.Transaction = transaction`). - Use

`Commit()` to save changes or `Rollback()` to undo on error. - Transactions help ensure data consistency and integrity. - For async code, use `BeginTransactionAsync()`, `CommitAsync()`, and `RollbackAsync()`.

10.5 ORM - Dapper

```
dotnet add package Dapper
```

```
using Microsoft.Data.SqlClient;
using Dapper;

await cn.OpenAsync();
var shows = await cn.QueryAsync<TvShow>("select * from TvShows");

public class TvShow
{
    public long Id {get; init;}
    public string ShowName {get; init;}
    public int ShowLength {get; init;}
    public string Summary {get; init;}
    public decimal Rating {get; init;}
    public string Episode {get; init;}
    public string ParentalGuide {get; init;}
}
```

10.6 ORM - Entity Framework

Entity Framework Core (EF Core) is a modern, open-source, object-database mapper for .NET. It enables developers to work with databases using .NET objects, eliminating most of the data-access code typically required. EF Core supports LINQ queries, change tracking, updates, and schema migrations across multiple database providers.

Example: Basic Usage with a DbContext and Model

```
using Microsoft.EntityFrameworkCore;

public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public decimal Rating { get; set; }
}

public class AppDbContext : DbContext
{
    public DbSet<TvShow> TvShows { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlServer("YourConnectionStringHere");
}

// Usage
using var db = new AppDbContext();
db.TvShows.Add(new TvShow { ShowName = "Friends", Rating = 4.8m });
db.SaveChanges();
```

```
var highRated = db.TvShows.Where(t => t.Rating > 4.0m).ToList();
```

10.6.1 Entity Framework: Raw SQL Queries with FromSql and FromSqlInterpolated

Entity Framework Core allows you to execute SQL queries using the `FromSql` and `FromSqlInterpolated` methods. These methods are safe against SQL injection because they always treat parameter values as SQL parameters, not as part of the SQL command text.

Example: Using FromSql with Parameters

```
using Microsoft.EntityFrameworkCore;

var minRating = 4.0m;
var shows = db.TvShows
    .FromSql($"SELECT * FROM TvShows WHERE Rating > {minRating}")
    .ToList();
```

Example: Using FromSqlInterpolated

```
var showName = "Friends";
var result = db.TvShows
    .FromSqlInterpolated($"SELECT * FROM TvShows WHERE ShowName = {showName}")
    .ToList();
```

Note:

- These methods can only be used on queries that return entity types (not arbitrary projections).

For more details, see the [official documentation](#).

10.6.2 Entity Framework Core: Disabling Change Tracking

By default, EF Core tracks changes to entities for automatic updates. For read-only scenarios, you can disable change tracking to improve performance using `.AsNoTracking()`.

Example:

```
using Microsoft.EntityFrameworkCore;

public class TvShow
{
    public int Id { get; set; }
    public string ShowName { get; set; }
    public decimal Rating { get; set; }
}

public class AppDbContext : DbContext
{
    public DbSet<TvShow> TvShows { get; set; }

    protected override void OnConfiguring(DbContextOptionsBuilder options)
        => options.UseSqlServer("YourConnectionStringHere");
}

// Usage: Query with change tracking disabled
using var db = new AppDbContext();
var shows = db.TvShows
```

```
.AsNoTracking()
.Where(t => t.Rating > 4.0m)
.ToList();
```

Use `.AsNoTracking()` for queries where you do not intend to update the returned entities.

10.7 Database Migrations - Entity Framework Core

Entity Framework Core supports code-based migrations to evolve your database schema alongside your models. Migrations are tracked in code and can be applied to the database as needed.

10.7.1 1. Add EF Core Tools

Install the EF Core CLI tools if not already present:

```
dotnet tool install --global dotnet-ef
```

Add the EF Core packages to your project:

```
dotnet add package Microsoft.EntityFrameworkCore
dotnet add package Microsoft.EntityFrameworkCore.Design
```

10.7.2 2. Create a Migration

After defining or updating your `DbContext` and models, create a migration:

```
dotnet ef migrations add InitialCreate
```

This generates a migration file in the `Migrations` folder.

10.7.3 3. Apply the Migration

Update the database to apply the migration:

```
dotnet ef database update
```

10.7.4 4. Example Migration Class

A generated migration might look like:

```
public partial class InitialCreate : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.CreateTable(
            name: "TvShows",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:Identity", "1, 1"),
                ShowName = table.Column<string>(nullable: false),
                Rating = table.Column<decimal>(type: "decimal(18,2)", nullable: false)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_TvShows", x => x.Id);
            }
        }
    }
}
```

```

        });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(name: "TvShows");
    }
}

```

10.7.5 5. Further Changes

To modify the schema, update your models and repeat the `dotnet ef migrations add` and `dotnet ef database update` steps.

For more, see the [official EF Core migrations documentation](#).

10.8 Database Migration - FluentMigrator

FluentMigrator is a migration framework for .NET that enables you to define database schema changes in C# using a fluent, expressive API. It supports versioned migrations, rollbacks, and can execute both fluent and raw SQL commands.

10.8.1 Example: Creating a Table with Fluent Syntax

```

using FluentMigrator;

[Migration(2023040701)]
public class CreateTvShowsTable : Migration
{
    public override void Up()
    {
        Create.Table("TvShows")
            .WithColumn("Id").AsInt64().PrimaryKey().Identity()
            .WithColumn("ShowName").AsString(50).NotNullable()
            .WithColumn("ShowLength").AsInt32().NotNullable()
            .WithColumn("Rating").AsDecimal(18,2).Nullable();
    }

    public override void Down()
    {
        Delete.Table("TvShows");
    }
}

```

10.8.2 Example: Executing Raw SQL in a Migration

```

using FluentMigrator;

[Migration(2023040702)]
public class InsertSampleData : Migration
{
    public override void Up()
    {

```

```

        Execute.Sql("INSERT INTO TvShows (ShowName, ShowLength, Rating) VALUES ('Friends', 1380, 4.5)");
    }

    public override void Down()
    {
        Execute.Sql("DELETE FROM TvShows WHERE ShowName = 'Friends'");
    }
}

```

10.8.3 Running Migrations

To run migrations, use the FluentMigrator CLI or integrate it into your build pipeline:

```
dotnet tool install -g FluentMigrator.DotNet.Cli
```

```
fluentmigrator migrate --assembly path/to/Your.Migrations.dll --provider sqlserver --connection "Se
```

10.9 Transactions and Isolation Levels

Transaction isolation levels determine how and when the changes made by one transaction become visible to other concurrent transactions. They help balance data consistency with system performance and concurrency.

10.9.1 Common Isolation Levels

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads	Supported By
Read Uncommitted	Yes	Yes	Yes	SQL Server, SQLite
Read Committed	No	Yes	Yes	SQL Server, PostgreSQL, SQLite*
Repeatable Read	No	No	Yes	SQL Server, PostgreSQL
Serializable	No	No	No	SQL Server, PostgreSQL, SQLite
Snapshot	No	No	No*	SQL Server, PostgreSQL†

* SQLite uses a simplified model; see notes below.

† PostgreSQL implements snapshot isolation as its default for **REPEATABLE READ**.

10.9.2 Isolation Level Descriptions

- **Read Uncommitted:** Allows reading uncommitted changes (“dirty reads”) from other transactions. Fastest, but least safe.
- **Read Committed:** Only reads data that has been committed. Prevents dirty reads, but non-repeatable and phantom reads are possible. Default in SQL Server and PostgreSQL.
- **Repeatable Read:** Ensures that if a row is read twice in the same transaction, it will not change. Prevents dirty and non-repeatable reads, but phantom reads can still occur.
- **Serializable:** Highest isolation; transactions are completely isolated from each other. Prevents dirty, non-repeatable, and phantom reads. May reduce concurrency.
- **Snapshot:** Each transaction sees a snapshot of the data as it was at the start of the transaction. Prevents dirty and non-repeatable reads, and usually phantom reads.

10.9.3 Example: Setting Isolation Level in SQL

SQL Server:

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN TRANSACTION;

SELECT * FROM TvShows WHERE Rating > 4.0;

-- ... do work ...

COMMIT TRANSACTION;
```

PostgreSQL:

```
BEGIN TRANSACTION ISOLATION LEVEL SERIALIZABLE;

SELECT * FROM tv_shows WHERE rating > 4.0;

-- ... do work ...

COMMIT;
```

SQLite: SQLite supports DEFERRED, IMMEDIATE, and EXCLUSIVE transactions, but you can simulate isolation levels:

```
BEGIN IMMEDIATE TRANSACTION;

SELECT * FROM TvShows WHERE Rating > 4.0;

-- ... do work ...

COMMIT;
```

- By default, SQLite is closest to SERIALIZABLE, but with some caveats due to its file-based locking.

10.9.4 Example: Setting Isolation Level in C

SQL Server:

```
using (var conn = new SqlConnection(connectionString))
{
    conn.Open();
    using (var tran = conn.BeginTransaction(System.Data.IsolationLevel.Serializable))
    {
        // All commands here use the specified isolation level
        // ...
        tran.Commit();
    }
}
```

PostgreSQL:

```
using (var conn = new NpgsqlConnection(connectionString))
{
    conn.Open();
    using (var tran = conn.BeginTransaction(System.Data.IsolationLevel.RepeatableRead))
    {
```

```

        // All commands here use the specified isolation level
        // ...
        tran.Commit();
    }
}

```

SQLite:

```

using (var conn = new SqlConnection(connectionString))
{
    conn.Open();
    using (var tran = conn.BeginTransaction(System.Data.IsolationLevel.Serializable))
    {
        // All commands here use the specified isolation level
        // ...
        tran.Commit();
    }
}

```

Note: SQLite only supports `Serializable` and `Read Uncommitted` isolation levels. `Read Committed` is emulated by default.

10.9.5 Summary Table

Isolation Level	Dirty Reads	Non-Repeatable Reads	Phantom Reads	SQL Server	PostgreSQL	SQLite
Read Uncommitted	Yes	Yes	Yes	Yes	No	Yes
Read Committed	No	Yes	Yes	Yes	Yes*	Emulated
Repeatable Read	No	No	Yes	Yes	Yes	No
Serializable	No	No	No	Yes	Yes	Yes
Snapshot	No	No	No*	Yes	Yes†	No

* PostgreSQL's default is `Read Committed`, but its `Repeatable Read` is implemented as snapshot isolation.

† PostgreSQL's `Repeatable Read` is snapshot isolation; true `Serializable` is stricter.

Tip: Choose the lowest isolation level that meets your consistency requirements to maximize performance and concurrency.

10.10 SQL Database Backup

Use `SqlConnection` and `SqlCommand` to create a bak copy only backup of a database.

```

using Microsoft.Data.SqlClient;

public async Task Backup(string connection, string saveFile,
    TimeSpan timeout)
{
    string backupDir = System.IO.Path.GetDirectoryName(saveFile);
    if (System.IO.Directory.Exists(backupDir) == false)
    {

```

```
        System.IO.Directory.CreateDirectory(backupDir);
    }

    if (System.IO.File.Exists(saveFile)){
        System.IO.File.Delete(saveFile);
    }

    var csb = new SqlConnectionStringBuilder(connection);
    string database = csb.InitialCatalog;

    var sql = @"
        BACKUP DATABASE [{database}]
        TO DISK = '{saveFile}'
        WITH FORMAT, COMPRESSION,
            MEDIANAME = '{database}-Data',
            NAME = 'Full Backup of {database}',
            COPY_ONLY;
    ";

    using var cn = new SqlConnection(connection);
    using var cmd = new SqlCommand();

    // CommandTimeout is an int measured in seconds
    cmd.CommandTimeout = (int)timeout.TotalSeconds;
    await cn.OpenAsync();
    cmd.CommandText = sql;
    cmd.Connection = cn;

    await cmd.ExecuteNonQueryAsync();
}
```

Part IV

Web and Hosting

Chapter 11

ASP.NET Core

ASP.NET Core is the cross platform web framework for .NET. The same runtime serves web pages, json APIs, and background services, and it runs on linux, windows, and mac.

Every ASP.NET Core application starts from a `Program.cs` that builds a host, registers services, configures the request pipeline, and runs.

```
var builder = WebApplication.CreateBuilder(args);

// 1. register services
builder.Services.AddControllersWithViews();

var app = builder.Build();

// 2. configure the middleware pipeline
app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseRouting();
app.UseAuthorization();
app.MapDefaultControllerRoute();

// 3. run
app.Run();
```

Middleware order is significant. Each `Use...` call wraps the ones after it, so authentication must be registered before authorization, and routing before either. Getting the order wrong produces confusing behaviour rather than a compile error.

11.1 Dependency Injection

[Structuring an Application](#) covers the concepts. ASP.NET Core builds them in: the container is already there, and the framework resolves your controllers, pages, and hosted services through it.

Register the repository and business classes from the **Repository Pattern** section on `builder.Services`.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<MajorSilence.DataAccess.ITestRepo>(sp =>
    new MajorSilence.DataAccess.TestRepo(
        builder.Configuration.GetConnectionString("Default")));
```

```
builder.Services.AddScoped<MajorSilence.BusinessStuff.TestStuff>();

builder.Services.AddControllers();

var app = builder.Build();
app.MapControllers();
app.Run();
```

A controller then declares what it needs in its constructor and the framework supplies it.

```
[ApiController]
[Route("api/[controller]")]
public class ShowsController : ControllerBase
{
    private readonly MajorSilence.BusinessStuff.TestStuff _stuff;

    public ShowsController(MajorSilence.BusinessStuff.TestStuff stuff)
    {
        _stuff = stuff;
    }

    [HttpGet]
    public IActionResult Get()
    {
        _stuff.DoStuff();
        return Ok();
    }
}
```

Scoped is the right lifetime for anything touching a database, because a scope is one http request. Two classes used in the same request share the connection and transaction; a later request gets fresh ones.

Do not resolve services by calling `provider.GetRequiredService` from inside your own code. That is the service locator pattern, and it hides dependencies that a constructor would have made obvious.

Connection strings belong in configuration rather than in code. `appsettings.json` holds the development value, and environment variables or a secret store override it in production.

```
{
  "ConnectionStrings": {
    "Default": "Server=localhost;Database=SqlPlayground;Trusted_Connection=True;"
  }
}
```

11.2 MVC

MVC splits a request into three parts: a **model** holding the data, a **view** rendering it, and a **controller** deciding what happens. It suits applications that serve rendered html pages.

Create a project.

```
dotnet new mvc -o YourWebApp
```

The rest of this chapter uses an `IShowRepo`, which is the same idea as chapter 3's `ITestRepo` but with methods that return shows: `GetShowsAsync`, `GetShowAsync`, `InsertAsync` and `DeleteAsync`.

A controller action returns a view along with the model it should render.

```

using Microsoft.AspNetCore.Mvc;

public class ShowsController : Controller
{
    private readonly IShowRepo _repo;

    public ShowsController(IShowRepo repo)
    {
        _repo = repo;
    }

    // GET /Shows
    public async Task<IActionResult> Index()
    {
        var shows = await _repo.GetShowsAsync();
        return View(shows);
    }

    // POST /Shows/Create
    [HttpPost]
    [ValidateAntiForgeryToken]
    public async Task<IActionResult> Create(TvShow show)
    {
        if (!ModelState.IsValid)
        {
            return View(show);
        }

        await _repo.InsertAsync(show);
        return RedirectToAction(nameof(Index));
    }
}

```

By convention the view for `Index` lives at `Views/Shows/Index.cshtml`. `@model` declares what the view was handed, and the razor syntax mixes c# into html.

```

@model IEnumerable<TvShow>

<h1>TV Shows</h1>

<table class="table">
    <thead>
        <tr><th>Name</th><th>Episode</th><th>Rating</th></tr>
    </thead>
    <tbody>
        @foreach (var show in Model)
        {
            <tr>
                <td>@show.ShowName</td>
                <td>@show.Episode</td>
                <td>@show.Rating</td>
            </tr>
        }
    </tbody>
</table>

```

```
</table>
```

Razor html encodes anything written with @ by default, so user supplied values cannot inject script. Only @Html.Raw bypasses that, which is why it should be rare and deliberate.

Validation attributes on the model drive both the client side and server side checks. ModelState.IsValid is the server side half and must never be skipped, because client side validation is trivially bypassed.

```
public class TvShow
{
    [Required]
    [StringLength(50)]
    public string ShowName { get; set; }

    [Range(0, 5)]
    public decimal Rating { get; set; }
}
```

Razor Pages is a lighter alternative that pairs each page with its own handler class instead of routing through controllers. For a site that is mostly pages rather than shared logic it is usually the simpler choice.

11.3 Minimal API

Minimal APIs express an http endpoint as a lambda, with no controller class. They suit small json services and are measurably faster to start.

```
dotnet new web -o YourApi
```

An entire service can fit in one file.

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.AddScoped<IShowRepo>(sp =>
    new ShowRepo(builder.Configuration.GetConnectionString("Default")));

var app = builder.Build();

app.MapGet("/shows", async (IShowRepo repo) =>
    Results.Ok(await repo.GetShowsAsync()));

app.MapGet("/shows/{id:long}", async (long id, IShowRepo repo) =>
{
    var show = await repo.GetShowAsync(id);
    return show is null ? Results.NotFound() : Results.Ok(show);
});

app.MapPost("/shows", async (TvShow show, IShowRepo repo) =>
{
    var id = await repo.InsertAsync(show);
    return Results.Created($"/shows/{id}", show);
});

app.Run();
```

Parameters are bound by source without any attributes: route values by name (id), registered services

from the container (`IShowRepo`), and a complex type from the json body (`TvShow`).

Group related endpoints so shared configuration is written once.

```
var shows = app.MapGroup("/shows").RequireAuthorization();

shows.MapGet("/", async (IShowRepo repo) => await repo.GetShowsAsync());
shows.MapDelete("/{id:long}", async (long id, IShowRepo repo) =>
{
    await repo.DeleteAsync(id);
    return Results.NoContent();
});
```

Minimal APIs and controllers can coexist in one application. Reach for controllers when the endpoint count grows, when filters and model binding conventions start being worth it, or when the team simply prefers the structure.

When they do coexist, keep them on separate path prefixes. Route matching is case insensitive, so a minimal API mapped at `/shows` and a `ShowsController` whose conventional route is `/Shows` are the same route. There is no error at startup; the minimal API simply wins and the controller action is never reached. Putting the json endpoints under `/api` avoids the whole class of problem.

Use controllers or Minimal APIs, but be consistent within a project. Mixing both for the same resource makes the routing hard to follow.

11.4 Blazor

Blazor builds interactive web UI in `c#` instead of javascript. Components are `.razor` files that combine markup, state, and event handlers.

```
dotnet new blazor -o YourBlazorApp
```

A component is a class with markup attached.

```
@page "/shows"
@inject IShowRepo Repo

<h1>TV Shows</h1>

@if (_shows is null)
{
    <p>Loading...</p>
}
else
{
    <ul>
        @foreach (var show in _shows)
        {
            <li>@show.ShowName (@show.Rating)</li>
        }
    </ul>
}

<input @bind="_newName" placeholder="Show name" />
<button @onclick="AddShow">Add</button>

@code {
```

```

private List<TvShow> _shows;
private string _newName = "";

protected override async Task OnInitializedAsync()
{
    _shows = (await Repo.GetShowsAsync()).ToList();
}

private async Task AddShow()
{
    if (string.IsNullOrEmpty(_newName))
    {
        return;
    }

    await Repo.InsertAsync(new TvShow { ShowName = _newName });
    _shows = (await Repo.GetShowsAsync()).ToList();
    _newName = "";
}
}

```

The part that decides everything else is the **render mode**, which controls where the component actually executes.

- **Static server rendering** - html is rendered once on the server and sent. No interactivity. Fastest, and the default in a new project.
- **Interactive Server** - the component runs on the server, and UI updates travel over a SignalR connection. Small download, but every user holds an open connection and all state lives in server memory.
- **Interactive WebAssembly** - the component runs in the browser on a .NET runtime. No connection to maintain and it works offline, at the cost of a larger initial download.
- **Interactive Auto** - server rendering on the first visit while the WebAssembly runtime downloads in the background, then WebAssembly afterwards.

Set the mode per component, so only the parts that need interactivity pay for it.

```
@rendermode InteractiveServer
```

The catch worth knowing up front: with WebAssembly, the component runs on the user's machine. It cannot open a database connection, and any secret it holds is readable. Components running in the browser must go through an http API, exactly as a javascript frontend would. The `@inject IShowRepo` shown above only works under server rendering.

Chapter 12

Containers, nginx and Kubernetes

Code that runs on your machine is not yet software anyone can use. This chapter is about the gap.

A container packages the application with the runtime it needs, so that what was tested is what ships. nginx sits in front of it as a reverse proxy, terminating TLS and serving static files, because Kestrel should not be the thing facing the internet directly. Kubernetes runs many containers across many machines and restarts them when they fail, which is worth its considerable complexity only once you actually have that problem.

Each step here adds capability and cost. Plenty of applications should stop after the first.

12.1 Containers - Docker

A Docker container is a lightweight, portable, and self-sufficient unit that packages an application and all its dependencies, ensuring consistent execution across different environments. Containers are isolated from each other and the host system, making deployment and scaling straightforward.

12.1.1 Example: Dockerfile for a .NET Web Application

```
# Use the official .NET SDK image for build
FROM mcr.microsoft.com/dotnet/sdk:10.0 AS build
WORKDIR /src

# Copy project files and restore first, so the restore layer is
# cached and only re-runs when a dependency actually changes.
COPY *.sln .
COPY YourWebApp/*.csproj ./YourWebApp/
RUN dotnet restore

COPY . .
RUN dotnet publish YourWebApp -c Release -o /app --no-restore

# Use the ASP.NET runtime image for hosting
FROM mcr.microsoft.com/dotnet/aspnet:10.0 AS runtime
WORKDIR /app
COPY --from=build /app ./
EXPOSE 8080
ENTRYPOINT ["dotnet", "YourWebApp.dll"]
```

Note the port. The official .NET container images run as a non root user and listen on 8080, not 80, since .NET 8. A non root user cannot bind a port below 1024.

12.1.2 Build and Publish with Docker Buildx and SBOM

one time setup

```
docker buildx create --use --name=buildkit-container --driver=docker-container
```

regular builds

```
# Build the image with SBOM (Software Bill of Materials) generation
docker buildx build --sbom=true -t yourusername/yourwebapp:latest .

# Publish (push) the image to a container registry (e.g., Docker Hub)
docker push yourusername/yourwebapp:latest
```

The `--sbom=true` flag generates a Software Bill of Materials, providing transparency into the components included in the image for improved security and compliance.

12.2 nginx

nginx is commonly placed in front of an ASP.NET Core application as a reverse proxy. Kestrel, the built in web server, is perfectly capable of serving traffic directly, but a proxy in front gives you TLS termination, a single entry point for several applications on one host, and static file serving without touching the runtime.

Install it on ubuntu.

```
sudo apt-get update
sudo apt-get install nginx
sudo systemctl enable --now nginx
```

Configure a site at `/etc/nginx/sites-available/yourapp`, then symlink it into `sites-enabled`.

```
server {
    listen 80;
    server_name yourapp.example.com;

    location / {
        proxy_pass          http://127.0.0.1:5000;
        proxy_http_version 1.1;

        # required for websockets, SignalR, and Blazor Server
        proxy_set_header Upgrade $http_upgrade;
        proxy_set_header Connection $connection_upgrade;

        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_cache_bypass $http_upgrade;
    }
}
```

`$connection_upgrade` is not built in and must be defined in the `http` block, usually in `/etc/nginx/nginx.conf`.

```
map $http_upgrade $connection_upgrade {
    default upgrade;
    ''      close;
}
```

Enable the site and reload. `nginx -t` checks the configuration before you apply it, which is worth doing every time.

```
sudo ln -s /etc/nginx/sites-available/yourapp /etc/nginx/sites-enabled/
sudo nginx -t
sudo systemctl reload nginx
```

12.2.1 Tell ASP.NET Core it is behind a proxy

Without this step the application sees every request as coming from 127.0.0.1 over plain http. Logging, rate limiting, and any redirect to https will all be wrong. `UseForwardedHeaders` must run before anything that depends on the scheme or client address.

```
dotnet add package Microsoft.AspNetCore.HttpOverrides
```

```
var builder = WebApplication.CreateBuilder(args);

builder.Services.Configure<ForwardedHeadersOptions>(options =>
{
    options.ForwardedHeaders =
        ForwardedHeaders.XForwardedFor | ForwardedHeaders.XForwardedProto;

    // Only trust the proxy in front of us. Clearing these lists
    // without setting KnownProxies would trust any caller's headers.
    options.KnownProxies.Add(System.Net.IPAddress.Parse("127.0.0.1"));
});

var app = builder.Build();

app.UseForwardedHeaders();
```

12.2.2 TLS with Let's Encrypt

Terminate TLS at nginx rather than in Kestrel. `certbot` obtains the certificate, edits the site configuration, and installs a renewal timer.

```
sudo apt-get install certbot python3-certbot-nginx
sudo certbot --nginx -d yourapp.example.com
```

12.2.3 Run the application as a service

`systemd` keeps the application running and restarts it after a crash or a reboot. Create `/etc/systemd/system/yourapp.service`

```
[Unit]
Description=Your ASP.NET Core application
After=network.target

[Service]
WorkingDirectory=/var/www/yourapp
ExecStart=/usr/bin/dotnet /var/www/yourapp/YourWebApp.dll
Restart=always
```

```
RestartSec=10
User=www-data
Environment=ASPNETCORE_ENVIRONMENT=Production
Environment=ASPNETCORE_URLS=http://127.0.0.1:5000
```

```
[Install]
WantedBy=multi-user.target
```

```
sudo systemctl daemon-reload
sudo systemctl enable --now yourapp
sudo systemctl status yourapp
```

Binding to 127.0.0.1 rather than 0.0.0.0 means the application is only reachable through nginx, not directly from the network.

12.3 Kubernetes

```
# use multiple kubeconfig files at the same time and view merged config
KUBECONFIG=~/.kube/config:~/.kube/kubconfig2
kubectl config view
kubectl config get-contexts
kubectl config current-context
kubectl get nodes
kubectl get namespaces
kubectl -n theNamespace get all
kubectl -n theNamespace get pods
kubectl -n theNamespace get deployments
kubectl -n theNamespace get service
kubectl -n theNamespace get ingress
kubectl -n theNamespace describe deployment theDeployment

# create a new pod yaml file. Edit the pod.yaml file to your needs.
kubectl run nginx --image=nginx --dry-run=client -o yaml > pod.yaml
kubectl create -f pod.yaml
```

Further reading:

- [kubectl cheat sheet](#)
- [Kubernetes Health Checks and Resource Reservations](#)

Chapter 13

JavaScript in the Browser

JavaScript is a lightweight, interpreted programming language primarily used for client-side web development. It enables dynamic content, user interaction, and DOM manipulation in browsers. This guide focuses on plain JavaScript for frontend tasks, avoiding dependencies and TypeScript for simplicity and maintainability.

13.1 Use htmx to Avoid Complicated JavaScript

Use [htmx](#) when you want to add dynamic, interactive features to your web application without writing or maintaining large amounts of custom JavaScript.

htmx gives you access to AJAX, CSS Transitions, WebSockets and Server Sent Events directly in HTML, using attributes, so you can build modern user interfaces with the simplicity and power of hypertext

13.2 fetch

Call a service using post with fetch api. These examples uses helper functions that are defined in [Helper functions](#) below.

13.2.1 call fetch - form-urlencoded

Use a custom `serialize` helper method to transform an javascript object (json) to an form url encoded format.

Example:

`?test_param=test value&another_param=another value`

```
function PostFormUrlEncoded(msg) {
  var data = serialize({
    test_param: "test value",
    another_param: "another value",
  });

  return fetch(site + "/some/url", {
    method: "POST",
    mode: "cors",
    headers: {
      "Content-Type": "application/x-www-form-urlencoded",
    },
  });
}
```

```

    },
    body: data,
  });
}

PostFormUrlEncoded("My comment")
  .then(status_helper)
  .then(json_helper)
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.log(error);
  });

```

13.2.2 call fetch - application/json

The content type **application/json** can use the builtin method **JSON.stringify** to send data.

```

const site = "https://example.com"; // Define your site URL

function PostJson(msg) {
  var data = JSON.stringify({
    test_param: "test value",
    another_param: "another value",
  });

  return fetch(site + "/some/url", {
    method: "POST",
    mode: "cors",
    headers: {
      "Content-Type": "application/json",
    },
    body: data,
  });
}

PostJson("My comment")
  .then(status_helper)
  .then(json_helper)
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.log(error);
  });

```

13.2.3 Helper functions

These helper functions implement some boiler plate code that will almost always be needed.

```

function status_helper(response) {
  if (response.status >= 200 && response.status < 300) {
    return Promise.resolve(response);
  }
}

```

```

    } else {
      return Promise.reject(new Error(response.statusText));
    }
  }

function json_helper(response) {
  return response.json();
}

function serialize(obj, prefix) {
  if (prefix === void 0) {
    prefix = null;
  }
  var str = [],
      p;
  for (p in obj) {
    if (Object.prototype.hasOwnProperty.call(obj, p)) {
      var k = prefix ? prefix + "[" + p + "]" : p,
          v = obj[p];
      str.push(
        v !== null && typeof v === "object"
          ? serialize(v, k)
          : encodeURIComponent(k) + "=" + encodeURIComponent(v)
      );
    }
  }
  return str.join("&");
}

```

13.3 async/await

Async and await support is built upon javascript promises. The following example is a slight modification on the PostJson example

Notice how the DownloadPage function is a GET and does not have a mode, headers, or body. A body must not be set on a GET but the other properties are settable. DownloadPage returns the response.text().

In contrast the PostJson function is a POST and sets the mode to cors, headers, and a body. PostJson returns the response.json().

```

async function DownloadPage(url) {
  const response = await fetch(url, {
    method: "GET",
  });

  if (response.status < 200 || response.status > 299) {
    throw new Error(response.status);
  }

  return response.text();
}

async function PostJson(url, msg) {
  var data = JSON.stringify({

```

```
    test_param: "test value",
    another_param: "another value",
  });

  const response = await fetch(url, {
    method: "POST",
    mode: "cors",
    headers: {
      "Content-Type": "application/json",
    },
    body: data,
  });

  if (response.status < 200 || response.status > 299) {
    throw new Error(response.status);
  }

  return response.json();
}

PostJson("https://majorsilence.com/non/existing/post/page", "My comment")
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.log(error);
  });

DownloadPage("https://majorsilence.com")
  .then(function (data) {
    console.log(data);
  })
  .catch(function (error) {
    console.log(error);
  });
```

13.4 jQuery

Avoid for new work.

13.5 Kendo UI

Avoid unless advanced controls are required.

Part V

Operations

Chapter 14

Monitoring and Observability

Effective monitoring is essential for Site Reliability Engineering (SRE) and DevOps teams to ensure the health, performance, and reliability of applications and infrastructure. Modern monitoring solutions provide real-time visibility, alerting, and analytics for both system-level and application-level metrics.

14.1 Prometheus Ecosystem

Prometheus is a leading open-source monitoring and alerting toolkit designed for reliability and scalability. It excels at collecting time-series metrics from targets via HTTP endpoints, supports flexible queries, and integrates seamlessly with cloud-native environments.

14.1.1 Key Features

- **Metrics Collection:** Scrapes metrics from applications, services, and infrastructure.
- **OpenTelemetry Support:** Integrates with **OpenTelemetry** for standardized observability data (metrics, traces, logs).
- **Blackbox Exporter:** Performs external health checks (HTTP, TCP, ICMP) to monitor service availability from the outside.
- **Alerting:** Built-in alert manager for notifications based on custom rules.

14.1.2 Service Discovery

Prometheus uses **service discovery** to automatically find and monitor targets (applications, services, or infrastructure) without manual configuration. This enables dynamic environments—such as Kubernetes, cloud platforms, or virtual machines—to be monitored as they scale up or down. Prometheus supports various service discovery mechanisms, including static configuration, DNS, file-based discovery, and integrations with cloud providers and orchestration systems.

Exporters are lightweight services that expose metrics from third-party systems (like databases, hardware, or messaging queues) in a format Prometheus can scrape. There are many official and community-supported exporters for popular technologies (e.g., `node_exporter` for system metrics, `blackbox_exporter` for endpoint probing, `mysqld_exporter` for MySQL).

Integrations refer to the broad ecosystem of tools and exporters that allow Prometheus to collect metrics from virtually any system, making it highly extensible and adaptable to diverse monitoring needs.

- **Prometheus Configuration**
- **Exporters and integrations**
- **Writing HTTP Service Discovery**
 - custom targets SD

- Use file-based service discovery to discover scrape targets
 - custom targets SD

14.1.3 Example: Exposing Metrics in .NET

Add the `prometheus-net` NuGet package to your ASP.NET Core app:

```
using Prometheus;

app.UseMetricServer(); // Exposes /metrics endpoint
app.UseHttpMetrics(); // Collects HTTP request metrics
```

Prometheus can then scrape metrics from `http://your-service/metrics`.

14.1.4 Example: Blackbox Exporter Configuration

Monitor an external HTTP endpoint:

```
# prometheus.yml
scrape_configs:
  - job_name: 'blackbox'
    metrics_path: /probe
    params:
      module: [http_2xx]
    static_configs:
      - targets:
        - https://your-service.example.com
    relabel_configs:
      - source_labels: [__address__]
        target_label: __param_target
      - target_label: __address__
        replacement: blackbox-exporter:9115
```

14.2 Grafana for Visualization

Grafana is a powerful open-source analytics and monitoring platform. It connects to Prometheus and other data sources to create interactive dashboards, visualizations, and alerts.

Ready to use dashboards for prometheus can be downloaded from [Grafana dashboards page](#).

14.2.1 Example: Prometheus Data Source in Grafana

1. Add Prometheus as a data source in Grafana (URL: `http://prometheus:9090`).
2. Create dashboards using queries like:

```
http_requests_total{job="myapp"}
up{job="blackbox"}
```

3. Set up alerts for key metrics (e.g., service downtime, high latency).

Chapter 15

Build Pipelines

Build pipelines are automated workflows that compile, test, and deploy code changes systematically. They ensure code quality, streamline development, and enhance reliability, integrating tasks like testing, deployment, and monitoring, resulting in efficient, error-free software delivery.

15.1 GitHub Actions

GitHub actions should go in the .github/workflows directory of a git project. The file type is yml but the name can be anything.

Example dotnet github action named dotnet.yml. This GitHub action builds a self contained dotnet console application, run tests, publishes the tests, and zips and archives the output artifacts for Windows, Linux, and Mac.

```
name: .NET

on:
  push:
    branches: [main]
  pull_request:
    branches: [main]

jobs:
  linux-build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: 10.0.x
      - name: Restore dependencies
        run: dotnet restore [YourSolution].sln
      - name: Build
        run: dotnet build [YourSolution].sln --no-restore -c Release
      - name: Test
        run: dotnet test -c Release [YourSolution].sln --verbosity normal --collect:"XPlat Code Cov
      - name: Test Report Publish
        uses: dorny/test-reporter@v2
```

```

    if: success() || failure() # run this step even if previous step failed
  with:
    name: unit tests
    path: **/*TestResults/*.trx
    reporter: dotnet-trx
- name: Publish
  run: dotnet publish [YourProject] -c Release -r linux-x64 -p:PublishReadyToRun=true --self-con
- name: Archive artifacts
  uses: actions/upload-artifact@v4
  with:
    name: [YourProject]-linux-x64
    path: |
      [YourProject]/bin/Release/net10.0/linux-x64
    retention-days: 1

windows-build:
  runs-on: windows-latest
  steps:
    - uses: actions/checkout@v4
    - name: Setup .NET
      uses: actions/setup-dotnet@v4
      with:
        dotnet-version: 10.0.x
    - name: Restore dependencies
      run: dotnet restore [YourSolution].sln
    - name: Build
      run: dotnet build [YourSolution].sln --no-restore -c Release
    - name: Publish
      run: dotnet publish [YourProject] -c Release -r win-x64 -p:PublishReadyToRun=true --self-conta
    - name: Archive artifacts
      uses: actions/upload-artifact@v4
      with:
        name: [YourProject]-win-x64
        path: |
          [YourProject]/bin/Release/net10.0/win-x64
        retention-days: 1

mac-build:
  runs-on: macos-latest
  steps:
    - uses: actions/checkout@v4
    - name: Setup .NET
      uses: actions/setup-dotnet@v4
      with:
        dotnet-version: 10.0.x
    - name: Restore dependencies
      run: dotnet restore [YourSolution].sln
    - name: Build
      run: dotnet build [YourSolution].sln --no-restore -c Release
    - name: Publish
      run: dotnet publish [YourProject] -c Release -r osx-x64 -p:PublishReadyToRun=true --self-conta
    - name: Archive artifacts
      uses: actions/upload-artifact@v4

```

```

with:
  name: [YourProject]-osx-x64
  path: |
    [YourProject]/bin/Release/net10.0/osx-x64
  retention-days: 1

```

15.1.1 GH Action to Create Linux Packages

```

jobs:
  linux-build:
    runs-on: ubuntu-latest
    env:
      SOLUTION_NAME: "YourSolution"
      DEVELOPER: "majorsilence"
      PROJECT: "Your Project"
      MAIN_EXE: "The Main Exe filename"
      PRODUCT: "product name"
      MAINTAINER: "Your Name <your@example.com>"
      VERSION: "1.0.0"
    steps:
      - uses: actions/checkout@v4
      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: 10.0.x
      - name: Build
        run: |
          dotnet restore ${env:SOLUTION_NAME}.sln
          dotnet build -c Release ${env:SOLUTION_NAME}.sln --no-restore
          dotnet publish -c Release -r linux-x64 --self-contained true
      - name: Prep for fpm
        run: |
          mkdir -p build/linux/opt/${env:DEVELOPER}/${env:PROJECT}
          cp -r ${env:PROJECT}/bin/Release/net10.0/linux-x64/publish/* build/linux/opt/${env:DEVELOPER}/${env:PROJECT}/${env:MAIN_EXE}
          chmod +x build/linux/opt/${env:DEVELOPER}/${env:PROJECT}/${env:MAIN_EXE}
          mkdir -p build/linux/usr/bin
          cat > build/linux/usr/bin/${env:DEVELOPER}-${env:PRODUCT} << 'EOF'
          #!/bin/sh
          /opt/${env:DEVELOPER}/${env:PROJECT}/${env:MAIN_EXE} "$@"
          rc=$?
          exit $rc
          EOF
          chmod +x build/linux/usr/bin/${env:DEVELOPER}-${env:PRODUCT}
      - name: Build deb package
        run: |
          cd build/linux
          fpm -s dir -t deb \
            --name ${env:DEVELOPER}-${env:PRODUCT} \
            --version ${env:VERSION} \
            --description "${env:DEVELOPER} ${env:PRODUCT} tool." \
            --maintainer "${env:DEVELOPER}" \
            --license "MIT" \
            --architecture all \

```

```
--deb-no-default-config-files \
--url "https://github.com/${ env.DEVELOPER }}/${ env.PROJECT }" \
--maintainer "${ env.MAINTAINER }" \
./
```

15.2 Jenkins

Find jenkins installation instructions at <https://www.jenkins.io/download/>.

Ubuntu Jenkins install

```
curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io.key | sudo tee \
/usr/share/keyrings/jenkins-keyring.asc > /dev/null
```

```
echo deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] \
https://pkg.jenkins.io/debian-stable binary/ | sudo tee \
/etc/apt/sources.list.d/jenkins.list > /dev/null
```

```
sudo apt-get update
```

```
sudo apt-get install jenkins openjdk-21-jdk-headless docker.io -y
```

```
sudo usermod -a -G docker jenkins
```

```
# java -jar jenkins-cli.jar -s http://localhost:8080/ install-plugin SOURCE ... [-deploy] [-name VAL]
```

15.2.1 Jenkins Plugin Setup

Install the docker pipelines and git branch source plugins

- <https://plugins.jenkins.io/docker-workflow/>
- <https://plugins.jenkins.io/github-branch-source/>
 - If github is being used

To display test results various Jenkin plugins are required.

- dotnet - [nunit](#)
- code coverage - [coverage](#)

15.2.2 Jenkins Dotnet Pipeline

Example of building and testing a dotnet project that has nunit testing enabled. If there is only one solution in the directory then the solution name does not need to be specified.

Save this file as **Jenkinsfile** in the projects base folder.

```
pipeline {
    agent none
    environment {
        DOTNET_CLI_HOME = "/tmp/DOTNET_CLI_HOME"
    }
    stages {
        stage('build and test') {
            agent {
                docker {
                    image 'mcr.microsoft.com/dotnet/sdk:10.0'
                }
            }
        }
    }
}
```

```
steps {
    echo "building"
    sh """
    dotnet restore [YourSolution].sln
    dotnet build [YourSolution].sln --no-restore
    dotnet test [YourSolution].sln --logger:"nunit"
    # for code coverage run the next line instead of the previous line
    # dotnet test -c Release [YourSolution].sln --collect:"XPlat Code Coverage" --logger:"nunit"
    """
}
post{
    always {
        nunit testResultsPattern: '**/TestResults/*.xml'
        recordCoverage(tools: [[parser: 'COBERTURA', pattern: '**/TestResults/**/*cobertura.xml']])
    }
}
}
```

See [Jenkins and pipelines](#), [Jenkinfile](#) for more examples.

Appendix A

Git Clients

Github new repo example.

```
mkdir your_repo
cd your_repo
echo "" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin git@github.com:YOUR_USERNAME/YOUR_REPO.git
git push -u origin main
```

Push an existing local repo to a new github repo.

```
git remote add origin git@github.com:YOUR_USERNAME/YOUR_REPO.git
git branch -M main
git push -u origin main
```

Git, show current branch.

```
git branch --show-current
```

Git, show remotes.

```
git branch --remotes
```

Git commit changes.

```
git commit -m "hello world"
```

Git pull/rebase from

```
git pull --rebase
```

Git pull from remote and branch.

```
git pull --rebase upstream main
```

A.1 Git Visual Studio

About Git in Visual Studio

A.2 Git Rider

How to efficiently use Git integration in JetBrains Rider

A.3 Tortoise Git

The Power of Git – in a Windows Shell

Tortoise Git - windows shell git integration

A.4 Github Desktop

Experience Git without the struggle

GitHub Desktop