

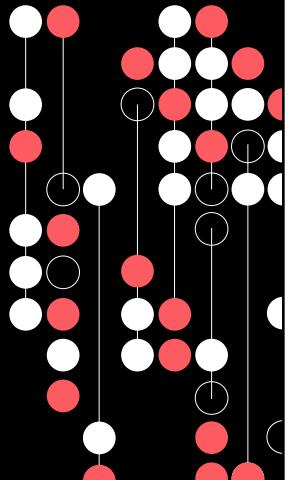


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Course Introduction



Session Objectives

- Course housekeeping
- Perform all the introductions

Session Content

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines



Housekeeping slide

Course Prerequisites

Essential skills

- Prior object-oriented programming experience
- At least 3 - 6 months working with C#
- Some familiarity with ASP .NET, ADO .NET and the Entity Framework
- Some familiarity with UML

Beneficial skills

- Background in OOP
- Web development (HTTP)
- Knowledge of JSON

NB This is a fast-paced course; there will be a certain time allowed for each lab and then we must move on to the next chapter.

Course Objectives

To enable you to use C# and .NET Core to:

- Adhere to best practice by using SOLID principles
- Recognise and use a number of commonly used coding patterns
- Understand the concepts of asynchronous programming and effectively use Tasks and the Task Parallel Library (TPL)
- Comprehend the concepts and principles behind Microservice Architectures and apply them in the creation of C# and .NET Core API web services
- Know how to create and build RESTful APIs using ASP.NET Core and to implement communication between microservices using message brokers
- Follow containerization principles using Docker to create and manage containers and services and also progressing to deployment and orchestration using Kubernetes.
- Be able to effectively test and debug distributed systems.
- Understand and apply security best practices for Microservices and how to implement scalability and load balancing strategies

Course Outline

- SOLID Principles
- Design Patterns
- Asynchronous Programming and Concurrency
- Introduction to ASP.NET Core Web API MVC
- ASP.NET MVC API Dependency Injection
- The Entity Framework
- Controllers and Actions
- Minimal API
- Microservices
- Containerisation with Docker
- Event Bus Message Brokers
- Deployment and Orchestration using Kubernetes
- Security and Scalability in Microservices



Activity: Introductions

Preferred name

Organisation & role

Experience of C# and OOP

Key learning objective

Hobby/Outside interest



Questions

Golden rule

- 'There is no such thing as a stupid question'

First amendment to the golden rule

- '... even when asked by an instructor'
- Please have a go at answering questions

Corollary to the golden rule

- 'A question never resides in a single mind'
- By asking a question, you're helping everybody



Allocation of Virtual Machines

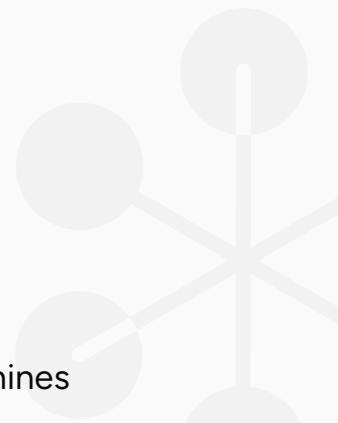
A number of labs make use of some hefty pieces of software which eat up processor power and memory.

For this reason, your tutor will allocate you a virtual machine which you are expected to make use of.

The virtual machine has all the necessary software that you need for the course already installed on it.

Summary

- Course administration
- Pre-requisites
- Course objectives
- Course outline
- Introductions
- Questions
- Allocation of Virtual Machines



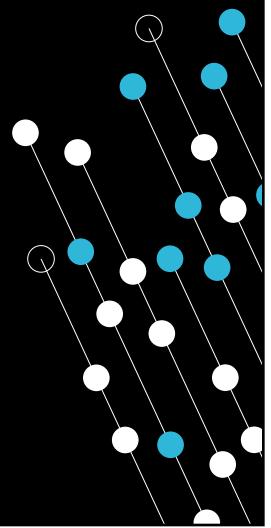


Advanced C#

Quick Start

Advanced C#

SOLID Principles

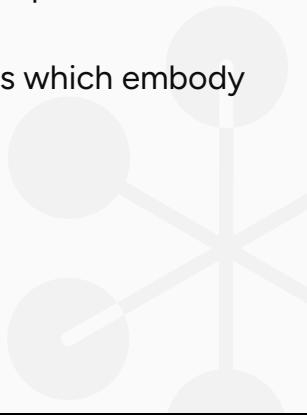


Session Objectives

Be reminded of programming best practice

To understand the main principles in refactoring classes

Leading into Design Patterns which embody those principles



Session Content

- C# Best Practice
- Experienced Developer best Practice
- Symptoms of a degrading design
- Background to SOLID Principles
 - Single-Responsibility Principle
 - Open/Closed Principle
 - Liskov Substitution Principle
 - Interface Segregation Principle
 - Dependency Inversion Principle

C# Best Practice Top 10! - Part 1

C# is a versatile and powerful programming language, and following best practices can significantly enhance code quality, maintainability, and readability. Here are the top 10 C# coding best practices:

1. Follow Naming Conventions
2. Keep Methods Short and Focused
3. Use Properties Instead of Public Variables
4. Utilize Exception Handling Properly
5. Apply the Principle of Least Privilege



1. Follow Naming Conventions:

- Use **PascalCase** for class names and method names.
- Use **camelCase** for local variables and method arguments.
- Use **UPPER_CASE** for constants.
- Names should be descriptive and clear about their purpose without being overly verbose.

2. Keep Methods Short and Focused:

- Each method should perform a single function. This makes methods easier to test, debug, and understand.

3. Use Properties Instead of Public Variables:

- Prefer properties with getters and setters to access fields of a class since properties provide better control over how values are set and retrieved.

4. Utilize Exception Handling Properly:

- Use try-catch blocks judiciously and avoid using exceptions for

control flow.

- Always catch specific exceptions rather than general exceptions where possible.

5. Apply the Principle of Least Privilege:

- Limit access levels by making class members private or protected unless they genuinely need to be public.

C# Best Practice Top 10! - Part 2

6. Avoid Magic Numbers and Strings
7. Document Your Code
8. Use Asynchronous Programming Wisely*
9. Practice Immutable Object Patterns When Appropriate
10. Adhere to SOLID Principles...**



6. Avoid Magic Numbers and Strings:

- Use named constants instead of hard-coding numbers or strings that appear in multiple places. This makes the code easier to maintain and understand.

7. Document Your Code:

- Use XML comments for documentation to provide summaries for classes, methods, and parameters. This documentation integrates with IDEs like Visual Studio and helps other developers understand your code quickly.

8. Use Asynchronous Programming Wisely*:

- Utilize `async` and `await` for asynchronous programming to keep UI responsive and improve performance for I/O-bound tasks.

9. Practice Immutable Object Patterns When Appropriate:

- When an object does not need to change after it is created, making

it immutable can reduce bugs and design complexity.

10. Adhere to SOLID Principles...**

* There's a session for this topic coming soon

**A detailed discussion based on SOLID Principles is coming up soon!

Experienced C# Developers Best Practice Top 10! -Part 1

For experienced C# developers looking to refine their craft further, best practices should emphasize advanced programming techniques, architectural design, and efficient resource management. Here are the top 10 best practices tailored for experienced C# developers:

1. **Leverage Advanced Language Features Wisely**
 - Like LINQ, extension methods, and expression-bodied member
2. **Optimize Asynchronous and Parallel Programming***
 - Utilize Task.WhenAll, Task.WhenAny, Parallel.For, and Parallel.ForEach to maximize performance
3. **Deep Dive into Memory Management**
 - Apply best practices in memory management, particularly with regard to IDisposable
4. **Implement Design Patterns and Principles Thoughtfully***
 - Apply design patterns where they are genuinely beneficial

- There's a session for this topic coming soon
1. Leverage Advanced Language Features Wisely:

Use features like LINQ, extension methods, and expression-bodied members to write concise and readable code. However, ensure that their use improves clarity and performance where applicable.
 2. Optimize Asynchronous and Parallel Programming*:

Go beyond basic async and await usage. Utilize Task.WhenAll, Task.WhenAny, Parallel.For, and Parallel.ForEach to maximize performance in multi-threaded and asynchronous scenarios, while ensuring proper exception handling and synchronization.
 3. Deep Dive into Memory Management:

Understand and apply best practices in memory management, particularly with regard to IDisposable, finalizers, and the effective use of using statements to manage resources efficiently.
 4. Implement Design Patterns and Principles Thoughtfully*:

Apply design patterns where they are genuinely beneficial, rather than fitting problems to patterns. Choose patterns that appropriately address your architectural needs and understand their trade-offs.

Experienced C# Developers Best Practice Top 10! - Part 2

5. Use Dependency Injection Effectively*

- Utilize Dependency Injection (DI) to decouple class dependencies

6. Emphasize on Code Testability and Robust Unit Tests

- Write testable code and build comprehensive unit

7. Adopt Advanced Architectural Styles

- To address complex business needs effectively.

8. Refactor Legacy Code With Strategy

- Develop a strategy for refactoring legacy systems. Introduce improvements incrementally and safely.

9. Integrate Secure Coding Practices*

- Prioritize security in the development lifecycle.

10. Stay Updated and Contribute to the Community

- Keep abreast of the latest C# features and .NET technologies. Engage with the community through forums, blogs, and conferences.

* There's a session for this topic coming soon

5. Use Dependency Injection Effectively*:

Utilize Dependency Injection (DI) to decouple class dependencies, facilitate easier unit testing, and manage class lifecycles, particularly in large applications or those built on frameworks like ASP.NET Core.

6. Emphasize on Code Testability and Robust Unit Tests:

Write testable code and build comprehensive unit tests using frameworks like NUnit or xUnit. Mock dependencies using tools like Moq or NSubstitute. Ensure that tests cover edge cases and failure modes, not just the happy paths.

7. Adopt Advanced Architectural Styles:

Depending on project requirements, explore and adopt architectural styles such as Clean Architecture, CQRS (Command Query Responsibility Segregation), or Event Sourcing to address complex business needs effectively.

8. Refactor Legacy Code With Strategy:

Develop a strategy for refactoring legacy systems. Introduce improvements incrementally and safely by employing techniques like strangling patterns, feature toggles, or branch by abstraction.

9. **Integrate Secure Coding Practices*:**

Prioritize security in the development lifecycle. Apply secure coding practices, such as validating input, using secure communication protocols, and managing data securely to prevent vulnerabilities like SQL injection, cross-site scripting, or data breaches.

10. **Stay Updated and Contribute to the Community:**

Keep abreast of the latest C# features and .NET technologies. Engage with the community through forums, blogs, and conferences. Share knowledge, contribute to open-source projects, and learn from peer reviews to continuously refine your skills.

Introduction to software architecture and dependencies

System architectures are multi-levelled

What causes an application to go 'bad' and become less stable?

At high level – feel the overall shape and structure

- Still a language independent view

At lower level – actual modules of code with interdependencies

- We can see design patterns, classes and components (our domain)

Original solution elegantly designed and implemented. However:

- Initial bug fixes/enhancements (hacks) keep a veneer of elegance
- Over time codebase degrades, ugly features appear
- More time and effort needed for even simple changes
- New re-design needed but moving goalposts, new design has V1.0 flaws

In this chapter we are going to consider software architecture at a few levels before deciding where as developer/implementers we can best affect the robustness of the code generated.

Architecture is multilevelled. At high levels there are the patterns that define the overall shape of an application and the structure related to the purpose of the application. At lower levels live the actual modules of compiled code with their interdependencies. This is where we can look at the nitty-gritty of design patterns, components and classes. We should recognise this is a huge area and we can provide a little knowledge and understanding, but often a little knowledge is dangerous. Much further reading is required, thousands of documents on the internet expand on the principles referred to in this chapter.

What is it that causes software to go awry, to become less stable? An original design may have been very elegant and clean and with no late design amendments to fit requirement changes. It was developed and made its way to a first release as V1.0. But at some variable speed the software slowly goes 'bad'. Initially a few hacked bug fixes and quick enhancements do not spoil the elegance of the original design. But over time it gets worse, possibly at an increasing rate until ugly features dominate the code. The code base becomes harder to maintain and before long the time/effort required to make the simplest of changes cause the owners and engineers to consider a complete redesign. This sort of redesign is often flawed as the designers are working against 'moving goalposts' as the original system is still evolving. There are then inherent problems latent in the new design before it even gets to its release date. When that happens – often after delays – the inherent problems surfacing in the new design mean people are already considering the next redesign/rebuild.

Symptoms of the design degrading

- Extensive literature (on the web) on this subject
- Four (partly related) symptoms are a clue that design has degraded
 - Rigidity – software difficult to change even in a simple way
 - Fragility (related) – tendency to break in many places when updated
 - Immobility – inability to reuse software components
 - Viscosity
- Multiple ways of making a change
- Easier to ‘hack’ a change than preserve design ('high' viscosity)
- Environment features can push engineers to non-optimal solution

The writers on this subject suggest four main symptoms that tell you when a design has gone bad. They are related to each other in a way that will become clearer through the chapter.

The symptoms are known as rigidity, fragility, immobility and viscosity.

These phrases were coined by a pioneering writer on OO Design, Robert C Martin in 2000 and now in general use.

Rigidity & Fragility

Rigidity

- Problematic to update in simple way
- Changes cascade to related changes in dependent routines
- Time and effort needed to follow a chain of repercussions
- Difficulty predicting duration of planned changes
- Owners become fearful of allowing non-critical changes
- Official rigidity sets in

Fragility

- Tendency to break in multiple places
- Failure can occur in areas with no conceptual relationship
- Fear increases of software always failing after an update
- Probability of further breakdowns increase

This is the tendency for software to be problematic to update even in a simple way. Changes seem to cause a cascade of little related changes in dependent routines. What starts out looking like a tiny amendment becomes a multi-day change routine affecting many modules of code as developers follow the repercussions through the application.

When this behaviour is exhibited, decision makers become fearful of allowing anything of a non-critical nature to be fixed.

This is driven from the fact that they find it harder to predict, how long a change will take.

The software design begins to become a big drain on developer resources. If these concerns become so acute that they refuse to allow changes to software, then official rigidity has set in.

Basically, a design deficiency has ended up being the cause of negative management policy.

Rigidity is related to fragility.

This is a tendency of the software to break in multiple locations each time it is updated. Failures can occur in areas that have no conceptual relationship with the area that was changed.

This sort of error fills the minds of decision makers and managers with dread.

Every time an update is authorised they live fear that the software will break in some unexpected way.

As the fragility increase, the probability of further breakages increase over time. This software becomes almost impossible to maintain.

Nearly every fix makes it worse, introducing more problems than are solved.

Such software causes customers and owners to suspect that engineers have lost control of their software. Distrust abounds, and credibility is lost.

Immobility & Viscosity

Immobility

- Inability to reuse components from other projects/parts of this one
- A similar module needed to another one already written
- It comes with too much baggage
- Work involved and risks in separating needed from unneeded
- Leads to rewrite and not reuse

Viscosity (aiming for 'low' viscosity)

- Design viscosity
 - Multiple ways of making a change work
 - One way preserves the design the other (easier) does not
 - So easier to do the wrong thing and harder to do the right thing
- Environment viscosity
 - Long re-build times, check-in check-out difficulties

Immobility is the inability to reuse software from other projects or from parts of the same project. An engineer will decide that he needs a module that is similar to one that another engineer wrote. However, it also often happens that the module in question has too much baggage that it depends upon.

After investigation the engineers discover that the work involved and risk taken to separate the wanted parts of the software from the unwanted parts is too great and so the software is simply rewritten instead of reused.

Viscosity comes in two forms: design viscosity and environment viscosity. When faced with a change, there is often more than one way to effect the change.

One way might preserve the design, another does not – it is a ‘hack’. When the design preserving method is harder to employ than the ‘hack’, then the design is said to have ‘high viscosity’. It becomes perhaps easier to do the wrong thing, but harder to do the right thing.

Viscosity of environment comes about when the development environment is inefficient.

If compile times are long, engineers will be tempted to make changes that don’t force large recompiles, even though those changes are non-optimal from a design standpoint. If the source code control system takes ages to check-in just a few files, then engineers will be tempted to make the changes that require as few check-ins as possible, regardless of whether or not the original design is maintained.

These symptoms are the signs of poor architecture. Any application that exhibits them is suffering from a design that is breaking apart from the inside out. But what causes this to take place?

Changing requirements

What has caused these degradations of design?

- Usually, requirements changing in ways that were unanticipated
- Changes made urgently by engineers unfamiliar with original design
- Change successful but design 'violated'
- Over time these accumulate

But software developers know that requirements will change

- So...
- Designs are partially at fault
- Need a way to make designs and code in them more resilient to change

The immediate cause of the degradation of the design is usually recognised as a case of 'requirements have been changing in ways that the initial design did not anticipate'. Often changes have to be made urgently, and perhaps by engineers who are not familiar with the original design philosophy. So, a change to the design might work, but 'violates' the original design.

Incrementally, as changes continue to be applied, these violations accumulate until the rot sets in.

But you cannot totally blame the drifting of the requirements for the degradation. As software engineers you know well enough that requirements inevitably change. Often the requirements document is one the most volatile documents in a project.

If designs are failing due to a constant demand of changing requirements, it is the designs that are (partially) at fault. A way is needed to make the designs more resilient to such changes and protect them from degrading.

Dependency management

- What sort of changes cause design degradation?
 - Usually changes that bring in new or unplanned ‘dependencies’
 - All symptoms just covered are caused by improper dependencies between modules
 - It is the dependency architecture that degrades
 - With it the ability to maintain the software
- To delay degradation, dependencies must be managed
 - ‘Dependency firewalls’ across which dependency does not propagate
- OO has principles and techniques (design patterns) to:
 - Build these firewalls
 - Manage dependencies
 - Maintain the dependency architecture

What are the sorts of changes that cause designs to degrade? Usually changes that introduce new and/or unplanned for dependencies. All the symptoms just covered are either directly or indirectly caused by improper dependencies between the various modules of the software. It is actually the dependency architecture that is degrading, and with it the ability to easily maintain the software.

In order to delay any degradation of the dependency architecture, the dependencies between the modules must be managed. This management consists of the creation of ‘dependency firewalls’. Across such firewalls, dependencies do not propagate.

Object Oriented Design has many principles and techniques for building such firewalls, and for managing module dependencies. It is these principles and techniques that will be discussed in the slides that follow.

First, we will examine the principles, and then the techniques (design patterns) that help maintain the dependency architecture of an application.

'SOLID' Principles

- Five Dependency management principles for OO Programming/Design:
 - Acronym 'SOLID' coined for these principles
 - Applied together more likely to create a more maintainable extendable application
 - Guidelines applied to software to remove 'code smells'
 - Programmer refactors source code to be legible and extensible
 - **S**ingle Responsibility principle (SRP)
 - **O**pen/Closed principle (OCP)
 - **L**iskov substitution principle (LSP)
 - **I**nterface segregation principle (ISP)
 - **D**ependency inversion principle (DIP)
- Part of overall strategy of Agile and Adaptive Programming

The *SOLID principles* are five dependency management principles used in OO programming and design. The acronym was introduced from the work of Robert Martin in early 2000's. Each letter represents another three-letter acronym that describes one principle.

When working with software in which dependency management is handled badly, code can become rigid, fragile and difficult to reuse. Code which is difficult to modify, either to change existing functionality or add new features, susceptible to the introduction of bugs, particularly those that appear in a when another area of code is changed. If you follow the SOLID principles, you produce code that is more flexible and robust, with a higher possibility for reuse.

The following slides give an overview of the five principles.

Single Responsibility Principle (SRP)

- A 'responsibility' is a 'reason to change'
- Should be one reason only for class to change
 - Should have single purpose
 - All methods related to primary function
 - Find multiple responsibilities
 - Greater likelihood of change being needed
 - Split into new classes
 - Represents a good way of identifying classes during the design phase
 - Reminds you to think of all the ways a class can evolve

The Single Responsibility principle says that there should never be more than one reason for a class to change. Effectively this means you should design your classes so that each has a single purpose. It certainly does not mean that each class should have a single method but that all of the methods and properties in the class are directly related to the class's primary function. Where a class has multiple responsibilities, these should be separated into new classes.

When a class has multiple responsibilities, the likelihood that it will need to be changed increases. Each time a class gets modified the risk of introducing a bug grows. By concentrating on a single responsibility, the risk is limited.

If a class does two things well then when it is only needed for one of those purposes then it comes with baggage.

SRP is a simple and fairly intuitive principle, but in practice it is often hard to get right as examples that follow will show.

Consider a class that compiles and prints a report, it could be changed for two reasons. The content of the report might change, the format could. These two things are changing for different reasons; one is 'substantive', and one 'cosmetic'. SRP says that these two aspects of the class are really two separate responsibilities, and should therefore be in separate classes or modules. What you are trying to do is avoiding the coupling of two things that change for different reasons at different times.

If there is a change to the report compilation process, there is greater danger that the printing code will break if it is part of the same class.

SRP Example

1 – violation (three reasons for change)

```
public class CO2Meter {
    public double CO2Saturation { get; set; }
    public void ReadCO2Level() {
        using (MeterStream ms = new Meterstream("co2")) {
            int raw = ms.ReadByte();
            CO2Saturation = (double)raw / 255 *100;// % calc
        }
    }

    public bool CO2High() { return CO2Saturation >= 2; }

    public void ShowHighCO2Alert() {
        Console.WriteLine(
            $"CO2 high ({CO2Saturation :F1}%)");
    }
}
```

Talks to hardware device to monitor CO2 levels

1) CO2 monitoring hardware could change

2) Process could change to consider a temperature factor

3) Alerting improved beyond 'Console' output

The code above is a class that communicates with a hardware device to monitor the CO2 levels in some fluid. The class includes a method named "ReadCO2Level" that retrieves a value from a stream generated by the CO2 monitoring hardware. It converts the value to a percentage and stores it in the CO2Saturation property. The second method, "CO2High", checks the CO2 saturation to ensure that it does not exceed the maximum level of 2%. The "ShowHighCO2Alert" shows a warning that contains the current saturation value. There are at least three reasons for change within the CO2Meter class. If the CO2 monitoring hardware is replaced the ReadCO2Level method will need to be updated. If the process for determining high CO2 is changed, perhaps to include a temperature variable, the class will need updating. Finally, if the alerting requirements become more sophisticated than outputting text to the console, the ShowHighCO2Alert method will need to be rewritten.

Gather together the things that change for the same reasons. Separate those things that change for different reasons.

This is really just a way to define cohesion and coupling. You want to increase the cohesion between things that change for the same reasons, and to decrease the coupling between those things that change for different reasons.



SRP EXAMPLE 1 – CODE REFACTORED TO THREE CLASSES

```
public class CO2Meter {  
    public double CO2Saturation { get; set; }  
    public void ReadCO2Level() {  
        using (MeterStream ms = new MeterStream("co2")) {  
            int raw = ms.ReadByte();  
            CO2Saturation = (double)raw / 255 * 100; // % calc  
        }  
    } // end class  
public class CO2SaturationChecker {  
    public bool CO2High(CO2Meter meter) {  
        return meter.CO2Saturation >= 2;  
    } // end class  
public class CO2Alerter {  
    public void ShowHighCO2Alert(CO2Meter meter) {  
        Console.WriteLine(  
            $"CO2 high ({meter.CO2Saturation :F1}%)");  
    } // end class
```

CO2Meter injected

Refactored code has functionality split into three classes. The first is the CO2Meter class. This retains the CO2Saturation property and the ReadCO2Level method. You could split these into separate classes but we will keep them together as they are closely related. The other methods are removed so that the only reason for change is replacement of the monitoring hardware.

The second class is named “CO2SaturationChecker”. This class includes a single method that compares the CO2 level with a maximum acceptable value. The method is the same as the original except for the addition of a parameter that injects a CO2Meter object containing the saturation level to test. The only reason for the class to change is if the test process is changed.

The final class is named “CO2Alerter”. This displays an alert that includes the current oxygen saturation level. Again, a CO2Meter dependency is injected. The one reason for the class to change is if the alerting system is updated.

Note – this refactored code breaks other SOLID principles in order that the application of the SRP is visible. Further refactoring of this example is necessary to achieve compliance with the other four principles.

Application of the SRP changes your code considerably. Classes in your projects

become smaller and cleaner. The number of classes present in a solution will increase accordingly, so it is important to organise them well using namespaces and project folders. The creation of classes that are tightly focused on a single purpose leads to code that is conceptually simpler to understand and maintain. Another benefit of having small, cohesive classes is that the chances of a class containing bugs drops. This reduces the need for changes, so the code is less fragile. As the classes perform only one duty, multiple classes will work together to achieve larger tasks. Along with the other principles this permits looser coupling. It can also make it easier to modify the overall software, either by extending existing classes or introducing new, interchangeable versions.

SRP EXAMPLE 2

```
public class Petrolstation {  
    public void OpenGate() {  
        // Open the gate if the time is later than 7 AM  
    }  
    public void Service(Vehicle vehicle) {  
        // Check if service station is opened and then  
        // complete the vehicle servicing  
    }  
    public void CloseGate() {  
        // Close the gate if the time has passed 7 PM  
    }  
}
```

Class has two responsibilities

- It is mixing 'opening and closing gate' functionality with providing core vehicle servicing role

SRP Example 2 after refactoring

```

public interface IGateUtility {
    void openGate();
    void closeGate();
}

public class PetrolStationUtility : IGateUtility {
    public void openGate() { /* Open gate if after 7AM */ }
    public void closeGate() { /* Close gate at 7PM */ }
}

public class PetrolStation {
    private IGateUtility gateutil;
    public PetrolStation(IGateUtility gateutil) {
        this.gateUtil = gateutil;
    }
    public void openForService() { gateutil.openGate(); }
    public void doService() {
        // If service station is open then service vehicle
    }
    public void closeForDay() { gateutil.closeGate(); }
}

```

Class is mixing 'opening and closing barrier' functionality with providing core vehicle servicing task

The refactored code above works as follows.

A new interface is defined that just contains 'gate' related utility methods. A concrete class implements the interface and supplies Open/CloseGate functionality.

The PetrolStation when instantiated is told of an IGateUtility object, he stores its reference (the assignment in the constructor). When it needs utility object at start/end of day it invokes the Open Close functionality that has been moved to a different class PetrolStationUtility.

If the way gates get opened/closed needs to change, or there is a new fangled barrier introduced, the PetrolStation will not need updating, it can just be passed a new concrete implementation of IGateUtility by client code.

Open / Closed Principle (OCP)

- OCP says make classes
 - ‘Open for extension’
 - New functionality added as new requirements generated
 - ‘Closed for modification’
 - Once developed, no modification except for bug fixes
 - Appears contradictory...
 - But achieved by referring to abstractions(interfaces typically) for dependencies rather than concrete classes
 - Interfaces fixed so classes depend on unchanging abstractions
 - Functionality added by new classes implementing the interfaces
 - Applying OCP limits the need to change source code once tested
 - Reduces the risk of introducing new bugs to existing code

The Open / Closed Principle (OCP) says that classes should be ‘open for extension’ but ‘closed for modification’. ‘Open to extension’ means that you design your classes so that new functionality can be added as new requirements are generated. ‘Closed for modification’ basically means that once you have developed a class you should never modify it, except to correct bugs.

On initial viewing the two parts appear to contradict each other. However, if you correctly structure your classes and their dependencies you can add functionality without editing existing source code.

This is normally achieved by referring to abstractions (interfaces or abstract classes) for dependencies, rather than using concrete classes.

Such interfaces can be fixed once developed so the classes that depend upon them can rely upon unchanging abstractions. Functionality is then added by creating new classes that implement the interfaces.

Applying the key Open Closed principle to your applications limits the need to change source code once it has been written, tested and debugged. It reduces the risk of introducing new bugs to existing code, leading to more robust software. Another side effect of the use of interfaces for dependencies is reduced coupling and increased flexibility.

Open / Closed Principle (OCP) – violation

```
public class Logger {  
    public void Log(string message, string loggingType) {  
        switch (loggingType) {  
            case "Console":  
                Console.WriteLine(message);  
                break;  
            case "File":  
                // Code to send message to default printer  
                break;  
        }  
    }  
}
```

Could be a (changeable)
enum?

Switch needs updating if
new 'loggingType'

The sample above is a basic module for logging messages. The Logger class has a single method that accepts a message to be logged and the type of logging to perform. The switch statement changes the action according to whether the program is outputting messages to the console or to the default printer. If you wished to add a third type of logging, perhaps sending the logged messages to a message queue or storing them in a database, you could not do so without modifying the existing code. Firstly, you would need to add new loggingType constants for the new methods of logging messages. Secondly, you would need to extend the switch statement to check for the new loggingTypes and output or store messages accordingly. This violates the OCP.

Open / Closed Principle (OCP) – code refactored

```

public class Logger {
    private IMessageLogger messageLogger;
    public Logger(IMessageLogger messageLogger) {
        this.messageLogger = messageLogger;
    }
    public void Log(string message) {
        messageLogger.Log(message);
    }
}
public interface IMessageLogger { void Log(string message); }
public class ConsoleLogger : IMessageLogger {
    public void Log(string message) {
        Console.WriteLine(message);
    }
}
public class PrinterLogger : IMessageLogger {
    public void Log(string message) {
        // Code to send message to printer
    }
}

```

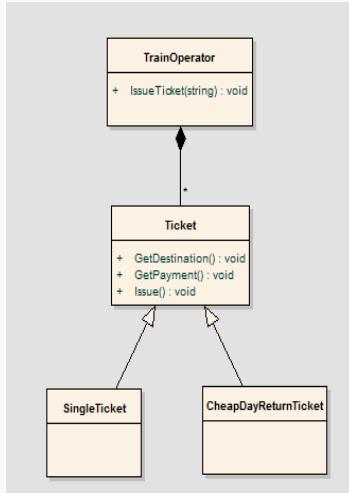
Dependency is only on type 'IMessageLogger'

Concrete classes that implement the dependent interface

We can easily refactor the logging code to achieve compliance with the OCP. Firstly, we need to remove any LoggingType enumeration, as this restricts the types of logging that can be included. Instead of passing the type to the Logger, we will create a new class for each type of message logger that we require. In the final code we will have two such classes, named "ConsoleLogger" and "PrinterLogger". Additional logging types could be added later without changing any existing code.

The Logger class still performs all logging but using one of the message logger classes described above to output a message. In order that the classes are not tightly coupled, each message logger type implements the IMessageLogger interface. The Logger class is never aware of the type of logging being used as its dependency is provided as an IMessageLogger instance using constructor injection.

OCP – Example 2 – separate the bits that change



```

class TrainOperator {
    public void IssueTicket
        (string type)
    {
        Ticket ticket = null;
        if (type == "Single")
            ticket= new SingleTicket();
        else if (type == "CheapDayRtn")
            ticket= new CheapDayRtnTicket(),
        ticket.GetDestination();
        ticket.GetPayment();
        ticket.Issue();
    }
}
  
```

The code snippet shows the `IssueTicket` method of the **TrainOperator** class. It creates a **Ticket** object based on the `type` parameter. A red oval highlights the conditional logic (`if` and `else if`) used to determine the ticket type. A callout box points to this highlighted area with the text: "This part changes with every new type of Ticket".

The if / else construct will need updating for every new sort of ticket. This needs to be factored (separated) out.

OCP – Example 2 - Closed For Modification

```
class TrainOperator {
    public void IssueTicket
        (string type)
    {
        Ticket ticket = null;

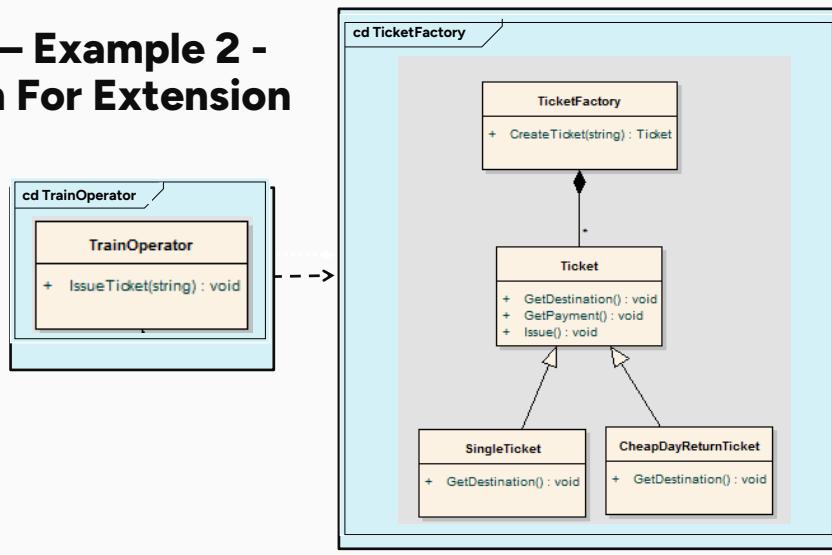
        ticket =
            TicketFactory
                .CreateTicket(type);

        ticket.GetDestination();
        ticket.GetPayment();
        ticket.Issue();
    }
}
```

```
public static class TicketFactory {
    public Ticket CreateTicket
        (string type) {
        Ticket ticket = null;
        if (type == "Single")
            ticket
                = new SingleTicket();
        else if (type == "CheapDayRtn")
            ticket
                = new CheapDayRtnTicket();
        return ticket;
    }
}
```

The CreateTicket() method is placed in a factory class method, the IssueTicket() method needs no more changes.

OCP – Example 2 - Open For Extension



TrainOperator now depends on the TicketFactory class whose CreateTicket method returns a base class 'Ticket' type.

Liskov Substitution Principle (LSP)

- LSP states:
 - If program code is using a base class reference, then the reference to object can be replaced with a reference to a derived class object without affecting the functionality of the program code
- Must ensure that any/all derived classes just extend without replacing functionality of base types
 - Otherwise, the new classes can produce undesired effects when they are used in existing program code
- Classic example of violation is surprisingly Rectangle and Square
 - You would think a Square 'is a' Rectangle and is fully substitutable
 - But not if Rectangle is mutable (see over)
- LSP says:
 - Will the client's perception that it is a Rectangle ever be broken if you pass in a Square object?

The whole point of LSP is to be able to pass around a subclass as the parent class without any problems.

This principle is just an extension of the Open Close Principle and it means that we must make sure that new derived classes are extending the base classes without changing their behavior.

The original statement by Barbara Liskov (1988) was:

"What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T."

It was then paraphrased later as:

"Methods that use references to base classes must be able to use objects of derived classes without knowing it."

The importance of this principle becomes obvious when you consider the consequences of violating it.

If there is a method which does not conform to the LSP, then that method uses a reference to a base class, but must *know about all the derivatives* of that base class. Such a method violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

Also anytime you see code that takes in some sort of baseclass or interface and then performs a check such as "if (someObject is SomeType)", there's a very good chance that that's an LSP violation.

Liskov Substitution Principle (LSP) – class definitions

```
class Rectangle {  
    public int Width { get; private set; }  
    public int Height { get; private set; }  
    public virtual void SetWidth(int w) { Width = w; }  
    public virtual void SetHeight(int h) { Height = h; }  
    public int Area { get {return Width * Height;} }  
}  
class Square : Rectangle {  
    public override void SetWidth(int w) {  
        base.SetWidth(w);  
        base.SetHeight(w); // Squares have identical width/height  
    }  
    public override void SetHeight(int h) {  
        base.SetWidth(h); // Squares have identical width/height  
        base.SetHeight(h);  
    }  
}
```

First clue that a Square should perhaps not inherit from Rectangle might be the fact that a Square does not need both Height and Width member variables!

SetWidth() of Rectangle must be virtual as the behaviour defined there is not suitable for a Square.

If you were able to do:

```
Square s = new Square();
```

s.setWidth(10) ...and it didn't change the Height as well it would not be a square anymore.

LSP is about following the contract of the base class.

Liskov Substitution Principle (LSP) – client code

- What would the user (who doesn't know how the factory produces a 'Rectangle') expect this code to print?

- 9, 15 or 25?

```
public void PlaywithRectangle() {
    Rectangle r = RectangleFactory();
    // Client code not certain what sort of Rectangle it has
    // But should not need to know!
    r.setHeight(3);
    r.setWidth(5);
    Console.WriteLine(r.Area);
}

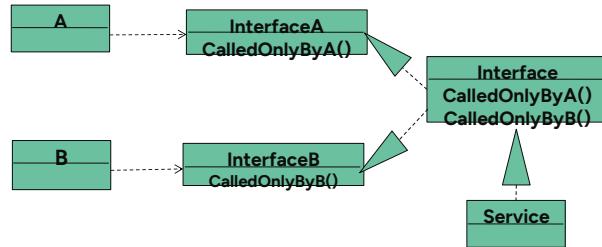
private Rectangle RectangleFactory() {
    ....
    return new Square();
}
```

The derived class has changed the behaviour of the base class – a clear violation.

For instance, although the C# compiler allows it, in a sub class you shouldn't throw a new exception in an overridden method if the base class wasn't built to expect it. The sub class will have changed the behavior. Client code that is treating an instance of the sub class as a base class object may get an unanticipated exception thrown at it. The same goes for if the base class throws ArgumentNullException if an argument is missing and the sub class allows the argument to be null, also a LSP violation.

Interface Segregation Principle (ISP)

- Clients should not be forced to depend on methods they do not use
 - If they are, then changes in one part of the system can cause changes in completely unrelated areas



A service that needs only one of the methods should not be dependent on an interface that demands two or more methods be implemented. The interface should be separated into two separate interfaces.

Interface Segregation Principle (ISP)

- ISP splits large interfaces into smaller and more specific ones
- Clients will only have to know of methods of interest to them
 - These 'shrunken' interfaces often referred to as *role interfaces*
- ISP helps to keep a system decoupled
 - Easier to re-factor, change and redeploy

The ISP principle was first used and formulated by Robert Martin in the 1990s while consulting at Xerox. Xerox had created a printer system that could perform a variety of tasks such as stapling and faxing. The software for this system was created from the ground up. As the software grew, making modifications became more and more difficult so that even the smallest change was incurring a significant redeployment time.

The design problem was that a single Job class was used by almost all of the tasks. Whenever a print job or a stapling job needed to be performed, a call was made to the Job class.

This Job class became very 'fat' with multitudes of methods specific to a variety of different clients. Because of this design, a staple job would know about all the methods of the print job, even though there was no use for them.

The solution Martin used to solve this problem utilised what is now known as the Interface Segregation Principle.

Applied to the Xerox problem, an interface layer between the Job class and its clients was added using the Dependency Inversion Principle (covered soon). Instead of having one large Job class, a Staple Job interface or a Print Job interface was created that would be used by the Staple or Print classes, respectively, calling methods of the Job class. Therefore, one interface was created for each job type, which were all implemented by the Job class.

Interface Segregation Principle (ISP) - violation

```
// interface segregation principle - bad example
interface IWorker {
    public void Work();
    public void Eat();
}

class Worker : IWorker {
    public void Work() { // ....working }
    public void Eat() { // ... eating in lunch break }
}

class Superworker : IWorker{
    public void Work() { //.... working much more }
    public void Eat() { //.... eating in lunch break }
}

class Manager {
    IWorker worker;
    public void Setworker(Iworker w) { worker = w; }
    public void Manage() { worker.Work(); }
}
```

What happens when a Robot (that Works but does not Eat) is introduced?

When we design an application, we take care how we abstract a module which contains several submodules. Considering code implemented by a class, we can have an abstraction of the system done in an interface. But if we want to extend our application, adding another module that contains only some of the submodules of the original system, we are forced to implement the full interface (and to write some dummy methods). Such an interface is named fat or polluted. Having interface pollution is not a good solution and often produces inappropriate behavior in the system.

ISP says clients should not be forced to implement interfaces they don't use. Instead of one fat interface many smaller interfaces are preferred based on groups of methods, each one serving one submodule.

Study the example above of a violation of ISP. We have a Manager class which represents the person which manages the workers. Two types of workers: some average, some very efficient. Both types of workers work and they need a daily lunch break. But then some robots start to work for the company as well, but they don't eat so they don't need a lunch break. But the new Robot class needs to implement the IWorker interface because robots work. Unfortunately, if the Robot class implements IWorker then what do we about the Eat method?

This is why in this case the IWorker is considered 'polluted'.

If you keep the present design, the new Robot class is forced to implement the Eat method. We can write a dummy method which does nothing (maybe a 1 second lunch break per day), but this can have subtle undesired effects in the application, for example the manager will have to implement the Eat method for the robot.

the IWorker interface should be split in two different interfaces.

Interface Segregation Principle (ISP) – solution

```
// interface segregation principle - reworked code
interface IWorkable { public void work(); }
interface IFeedable { public void Eat(); }
interface IWorker : IFeedable, IWorkable { }
class Worker : IWorker {
    public void work() { // working }
    public void Eat() { // ... eating in lunch break }
}
class Robot : IWorkable {
    public void work() { //.... working much more }
}
class Superworker : IWorker {
    public void work() { //.... working much more }
    public void Eat() { //.... eating in lunch break }
}
class Manager {
    IWorkable worker;
    public void Setworker(IWorkable w) { worker=w; }
    public void Manage() { worker.work(); }
}
```

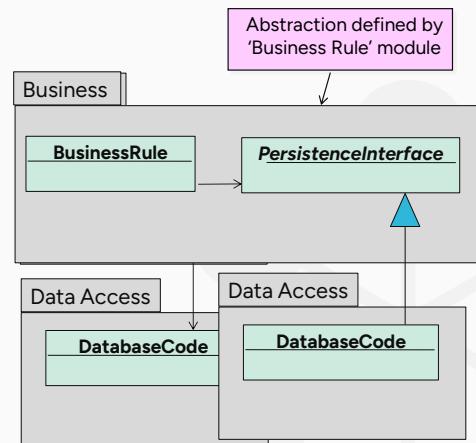
Here is the reworked code supporting the Interface Segregation Principle. By splitting the IWorker interface in two different interfaces the new Robot class is no longer forced to implement the Eat() method. Also if we need another functionality for the robot like recharging we create another interface IRechargeble with a method Recharge().

Like every principle, ISP is a principle which requires additional time and effort spent to apply it during the design time and it increases the complexity of code.

But the benefit is the creation of a flexible design. If you overdo it you will end up with a lot of interfaces with single methods So experience and common sense need to prevail when identifying the areas where extension of code is more likely to happen in the future.

Dependency Inversion Principle (DIP)

- “High-Level modules should not depend on low-level modules – both should depend on abstractions”
- “Abstractions should not depend on details. Details should depend on abstractions”



In conventional application architecture, lower-level components are designed to be consumed by higher-level components which enable increasingly complex systems to be built. In this composition, higher-level components depend directly upon lower-level components to achieve some task. This dependency upon lower-level components limits the reuse opportunities of the higher-level components.

The goal of the dependency inversion principle is to decouple application glue code from application logic. Reusing low-level components (application logic) becomes easier and maintainability is increased. This is facilitated by the separation of high-level components and low-level components into separate packages/libraries, where interfaces defining the behaviour/services required by the high-level component are owned by, and exist within, the high-level component's ‘package’. The implementation of the high-level component's interface by the low-level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship. Various patterns are then employed to facilitate the run-time provisioning of the chosen low-level component implementation to the high-level component.

Dependency Inversion Principle (DIP) – violation example

```
// Dependency Inversion Principle - Bad example
class Worker {
    public void work() { // ....working }
}
class Manager {
    Worker worker;
    public void SetWorker(Worker w) {
        worker = w;
    }
    public void Manage() {
        worker.Work();
    }
}
// now add ...
class SuperWorker {
    public void work() { //.... working much more }
}
```

This is an example which violates the Dependency Inversion Principle. We have a manager class which is a high-level class, and a low-level class called Worker. We might need to add new functionality to our application to model the changes in the company structure determined by the employment of new specialised workers. We might consider creating a new class SuperWorker for this.

The Manager class may be quite complex and now we have to change it in order to introduce the new SuperWorker. What are the disadvantages?

- We have to change the Manager class (it may be complex and will involve time and effort to make the changes)
- Some of the current functionality from the manager class might be affected.
- Significant unit testing will need to be redone

All these problems could take a lot of time to solve and they might introduce new errors in the old functionality. The situation would be different if the application had been designed following the Dependency Inversion Principle (see overleaf).

Dependency Inversion Principle (DIP) – solution

```
// Dependency Inversion Principle - Good example
interface Iworkable { public void work(); }
class Worker : Iworkable {
    public void work() { // ....working }
}
class Manager {
    Iworkable worker;
    public void Setworker(Iworkable w) {
        worker = w;
    }
    public void Manage() {
        worker.work();
    }
}
// now add ...
class Superworker : Iworkable {
    public void work() { //.... working much more }
}
```

We design the Manager class, an IWorkable interface and the Worker class implementing the IWorkable interface. If/when we add the SuperWorker class all we have to do is implement the IWorkable interface for it. No additional changes in the existing classes.

The code above supports the Dependency Inversion Principle.

In this new design the IWorkable Interface is the abstraction layer.

The abstraction defining the behaviour/services required by the high-level component are owned by, and would exist within the high-level component's 'package'. The implementation of the high-level component's interface by the low level component requires that the low-level component package depend upon the high-level component for compilation, thus inverting the conventional dependency relationship. When this principle is applied it means the high-level classes are not working directly with low-level classes, they are using interfaces as an abstract layer. In this case, instantiation of new low-level objects inside the high-level classes(if necessary) can not be done using the '**new**' operator. Instead, some of the Creational design patterns can be used, such as Factory Method, Abstract Factory or Prototype.

Now, the problems seen previously are solved (assuming there is no change in the high level logic):

- Manager class doesn't require changes when adding SuperWorkers.
- Minimised risk to affect old functionality present in Manager class since we don't change it.
- No need to redo the unit testing for Manager class

Of course, using this principle means an increase in effort, which will result in more classes and interfaces to maintain and slightly more complex code. However the benefit will be greater flexibility. The technique should **not** be applied blindly nor should it be used in every case. If class functionality is likely to remain unchanged in

Design Patterns embody principles

- They are a great way of grasping and implementing principles in your design/code
- You have to strike a balance between:



Too simple
Easier to hack than follow the design
Rigid/Fragile/Repetitive etc.



Needless Complexity

→ How do you find that balance? – next slide...



...by Unit Testing

Exposes software that ought to be de-coupled

We naturally put responsibilities together

- Software design is largely about separating the responsibilities that are subject to change
- You don't want to do it if the change never happens
- When you do have to do it, you want to do this just once
 - I.e. take the first bullet then refactor so the following bullets don't hurt
 - Never say "we'll come back and fix that later"
 - Agile design is a process, not an event

Unit Test is probably the best tool we have for driving good design

Common Design Patterns

- Façade
- Composite
- Command
- Observer (subsumed into .NET as events)
- Strategy
 - Defines a family of algorithms, encapsulates each algorithm, and makes the algorithms interchangeable within that family
- Factory
 - Creates objects without exposing instantiation logic to client
- Singleton
 - A class self-instantiates only instance

[More Later...](#)

Further study and experience needed.

Hands on Lab

Book Store Inventory System

Objective:

Your goal is to refactor code so that adheres to SOLID principles.

You are provided with a "Starter" program that manages an inventory system for a bookstore. The system allows adding books and CDs to the inventory and calculating the total stock value. The original code violates multiple SOLID principles. Your task is to refactor this code to make it more modular, maintainable, and extensible.

Review

- Symptoms of a degrading design
 - Rigidity, fragility, immobility, viscosity
- Degradation of dependency architecture
- SOLID
 - SRP / OCP / LSP / ISP / DIP



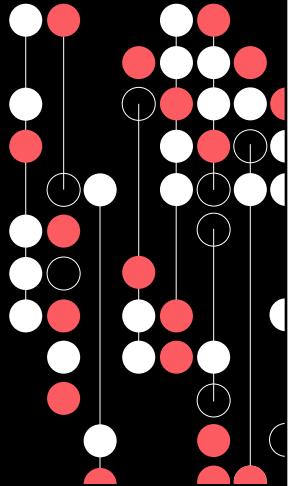


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Design Patterns



Session Objectives

Provide an introduction to coding design patterns. To understand their benefits and how to implement them and to gain exposure to a number of classic patterns.

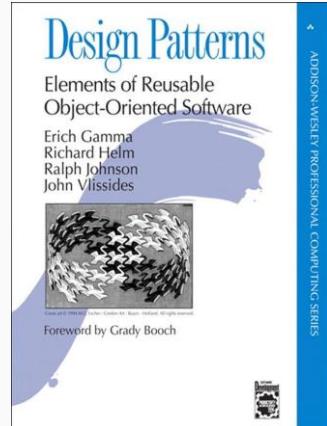
Session Content

- Introduction to Design Patterns
- Pattern Classifications
- Creational Patterns
 - Singleton, Factory, Builder, Prototype
- Structural Patterns
 - Adapter, Decorator, Composite, Facade
- Behavioural Patterns
 - Strategy, Observer, Command

Design Patterns - What Are They?

Common solutions to problems which are regularly encountered in programming.

Created and collated by the "Gang of Four" in "Design Patterns" book.



Design Patterns

Software design patterns are common solutions to problems which are regularly encountered in programming. These particular patterns deal with object-oriented programming exclusively. So, applying these patterns to any other programming paradigm is not a good idea. Some pattern proponents say that in the object-oriented world, these design patterns are full-fledged best practices. Others may not go quite as far as this, but all accept OO coding patterns are things that should be taken seriously. The patterns we'll be looking at originate from a book called, *Design Patterns, Elements of Reusable Object-oriented Software*, written by a group of authors who have come to be known as The Gang of Four or GOF.

Design Patterns - WHY?

Some of the benefits of using design patterns are:

- Save time.
- There are many C# and .NET design patterns that we can use in our C# .NET projects.
- Promotes reusability
- Help to reduce the total cost of ownership (TCO) of the software product
- Lead to faster development.

Design patterns are already defined and provides industry-standard approach to solve a recurring problem. So, it saves time if we sensibly use the design pattern.

There are many C# and .NET design patterns that we can use in our C# .NET projects.

Using design patterns promotes reusability, that leads to more robust and highly maintainable code.

Design patterns help to reduce the total cost of ownership (TCO) of the software product. Since design patterns are already defined, it makes our code easier to understand and debug.

Design patterns lead to faster development and new members of a team can understand them easily.

Pattern Classifications

- Creational
 - Singleton
 - Factory
 - Builder
 - Prototype
- Structural
 - Adapter
 - Decorator
 - Composite
 - Facade
- Behavioural
 - Strategy
 - Observer
 - Command



According to the GOF, design patterns are classified in three groups: Creational, Structural, Behavioural. For example, singleton, factory, builder and prototype are creational design patterns; adapter, decorator, composite and facade are structural patterns, and strategy, observer and command patterns are behavioural patterns. In the following slides, we will start examining design patterns by looking how they can be implemented in C# code.

Creational Patterns

- Deal with scenarios that involve the creation of an object or objects
 - tries to create them in a manner that is appropriate to the situation.
- Creating objects in a haphazard manner could result in:
 - Design problems
 - Over-complex design.
- Creational patterns aim to solve this by:
 - Separating a system from how its objects are created and composed.

Creational patterns deal with scenarios that involve the creation of an object or objects, trying to create them in a manner that is appropriate to the situation. By simply creating objects in a haphazard manner could result in design problems and/or an over-complex design. Creational patterns aim to solve this by separating a system from how its objects are created and composed.

Creational Pattern Classifications

- Singleton
- Factory
- Builder
- Prototype



The Singleton Pattern - Purpose

The purpose of the Singleton design pattern is to create only one instance of a class.

In other words, when an instance of a class is needed, if there is no instance, a new one is created. However, if an instance has already been created, the existing instance is used.

Singleton Pattern - Implementation

The implementation of the Singleton pattern involves several standard steps:

- Make the class constructor private
- Create a private static variable that holds the single instance of the class.
- Provide a public static method that returns the instance of the class.

Make the class constructor private to prevent other objects from using the new operator to create instances of the class.

Create a private static variable that holds the single instance of the class.

Provide a public static method that returns the instance of the class. This method often includes logic to create the instance if it hasn't been created yet, ensuring that the class is lazily instantiated.

Singleton Pattern Example Code



```
public sealed class Singleton {  
    private static Singleton instance = null;  
  
    private Singleton(){ }  
  
    public static Singleton Instance {  
        get {  
            if (instance == null)  
            {  
                instance = new Singleton();  
            }  
            return instance;  
        }  
    }  
}
```

Singleton Pattern Thread Safe Example

In a multi-threaded environment, there is a possibility two threads could evaluate the test (`if (instance == null)`) and find it to be true and then go on and create two instances.

Consequently, you may need to create a thread-safe version:

```
public sealed class Singleton {
    private static Singleton instance = null;
    private static readonly object padlock = new
    object();

    Singleton() { }

    public static Singleton Instance {
        get {
            lock (padlock) {
                if (instance == null) {
                    instance = new Singleton();
                }
                return instance;
            }
        }
    }
}
```

You may find the following article taken from the csharpindepth.com that takes a (much) deeper dive into thread safety, lazy type initialisation and performance: [Implementing the Singleton Pattern in C# \(csharpindepth.com\)](http://csharpindepth.com/Articles/General/Implementing_the_Singleton_Pattern_in_C_.aspx). However, do note the underlying discussion was last updated in 2011 so things may have changed.

DEMO: Singleton Pattern

The example demonstrates how any consumer of the Singleton class will receive the same instance of Singleton, ensuring that global state managed by the singleton is shared across its consumers. This pattern is very effective for operations where actions need to be controlled from a single point, like caching, thread pools, or windows registry operations.

Explanation of the Example

- The Singleton class has a private static variable instance which holds the only instance of the class.
- The constructor of the class is private, which prevents instantiation from outside of the class.
- The public static property Instance provides the global point of access to the singleton. The property ensures that the singleton instance is created only when it is first needed and subsequently returns the same instance for every subsequent request.
- The addition of the lock statement around the instantiation logic ensures that the class is thread-safe. This means if multiple threads try to get the instance at the same time, only one will be able to execute the instantiation code block, while the others will wait.

The Factory (Method) Pattern - Purpose

Delegate responsibility for creating an object to another

- Simplify construction
 - Dependencies may need to be initialised or switched
 - Give meaningful names to multiple constructors
- Abstract the exact product type
 - Product can be developed without impacting the client
- Facilitate Testing with dependency injection



The factory (method) pattern is a creational pattern in which interfaces are defined for creating families of related objects without specifying their actual implementations. The pattern provides an interface for creating objects in a base class but allows derived classes to alter the type of objects that will be created. The pattern is particularly useful for managing and centralizing the creation of objects, especially when the exact types or dependencies of the objects are not directly known to the client that uses them.

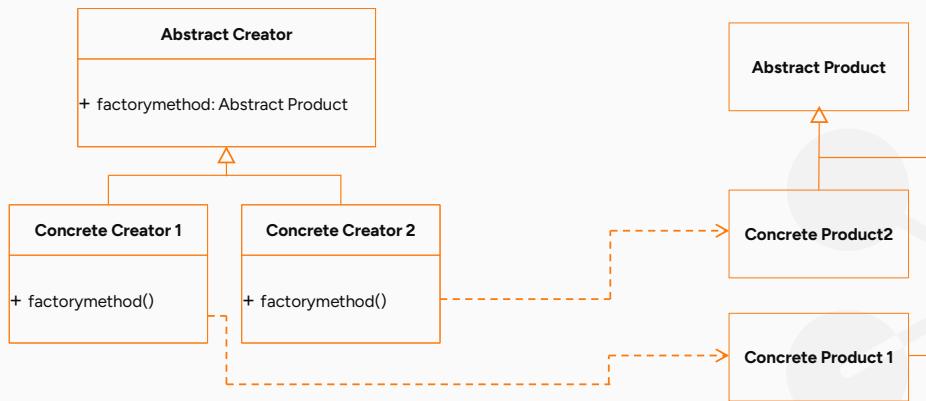
Factory Pattern - Implementation

- Components of the Factory Pattern
- Product: Defines the interface of objects the factory method creates.
- ConcreteProduct: Specific classes that implement the Product interface.
- Creator: Declares the factory method, which returns an object of type Product.
- ConcreteCreator: Overrides the factory method to return an instance of a ConcreteProduct.

The main idea is to use a factory method to deal with the problem of creating objects without specifying the exact class of object that will be created. . When using this pattern, you create factories which return many kinds of related objects. Basically, when using the factory pattern, we get rid of if-else blocks and, instead, have a factory class for each derived class, then, an abstract Factory class that will return the subclass based on the input factory class.

- Product: Defines the interface of objects the factory method creates.
- ConcreteProduct: Specific classes that implement the Product interface.
- Creator: Declares the factory method, which returns an object of type Product. Creator may also define a default implementation of the factory method that returns a default ConcreteProduct object.
- ConcreteCreator: Overrides the factory method to return an instance of a ConcreteProduct.

Factory Pattern - UML



The abstract creator supports an interface that can generate and return an abstract product. The specific products produced (Concrete Product 1 and Concrete Product 2) are variations of this abstract product.

In the concrete factory creator class, the factory method is implemented as a pure virtual function with a return type of Abstract Product, enabling the return of any specific product derived from this abstract base.

The decision of what specific product the factory method returns is left to the concrete creators.

Factory Pattern Example Code

```
// Abstract Creator Class  
public abstract class VehicleFactory {  
    // Factory Method  
    public abstract Vehicle CreateVehicle();  
  
    // Concrete Creator Class  
    public class CarFactory : VehicleFactory {  
        public override Vehicle CreateVehicle() {  
            return new Car();  
        }  
    }  
  
    // Abstract 'Product' Class  
    public abstract class Vehicle {  
        public abstract void Drive();  
    }  
  
    // 'Concrete 'Product' Class  
    public class Car : Vehicle {  
        public override void Drive() {  
            Console.WriteLine("Driving a car");  
        }  
    }  
}
```

Factory Code

Client Code

```
VehicleFactory carFactory = new CarFactory();  
Vehicle myCar = carFactory.CreateVehicle();  
myCar.Drive();  
  
VehicleFactory lorryFactory = new LorryFactory();  
Vehicle myLorry = lorryFactory.CreateVehicle();  
myLorry.Drive();
```

DEMO: Factory Pattern

This structure allows the program to defer instantiation logic to derived classes, making the code more modular and easier to introduce new "products" without changing existing client code. The client (main program) only deals with the VehicleFactory and Vehicle abstract classes/interfaces. It does not know the details of the specific types of vehicles created. This is ideal for scenarios where you have multiple related products that may vary significantly but share common interfaces or base classes.

Explanation of the Example

- Vehicle (Product) defines an interface for the type of object the factory method will create.
- Car and Lorry (ConcreteProducts) are different implementations of the Vehicle class.
- VehicleFactory (Creator) provides the factory method CreateVehicle that returns an object of type Vehicle. It is an abstract class.
- CarFactory and LorryFactory (ConcreteCreators) implement the CreateVehicle factory method to return instances of Car and Lorry, respectively.

The Abstract Factory Pattern

The Abstract Factory Pattern

- Offers an interface that allows families of related or dependent objects to be created without needing to specify their concrete classes.
- The pattern is used when you have interrelated objects that need to be created together.
- Uses a series of factory methods, one for each kind of product to be created

The Abstract Factory pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. The pattern is used when you have interrelated objects that need to be created together. The pattern is particularly useful when a system should be independent of how its products are created, composed, and represented. It involves a series of factory methods, one for each kind of product to be created.

The Factory vs. Abstract Factory Pattern

Are the Factory and Abstract Factory Pattern the Same?

In short, No though the Abstract Factory pattern does make use of the Factory pattern.

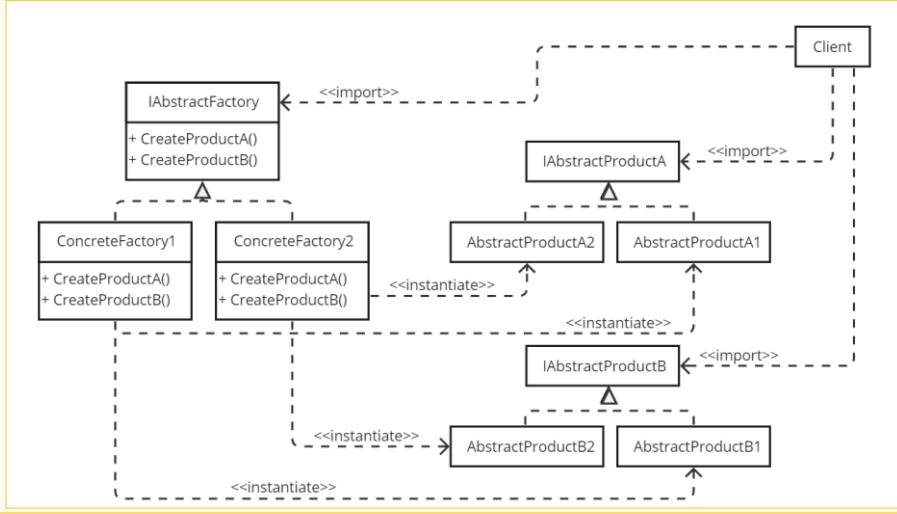
The Abstract Factory pattern allows each factory to create a suite of related products without having to understand their makeup. In contrast, the Factory Method only allows a factory to produce a single type of product

Intent and Complexity: The Factory Method is about creating one product but deferring the instantiation logic to subclasses. Whereas the Abstract Factory pattern is used for creating families of related or dependent products without having to specify/understand their concrete classes.

Implementation: The Factory Method involves a single method to create products, whereas The Abstract Factory pattern uses multiple factory methods to create related products.

Use Case: Use Factory Method when you want to extend a single product type's instantiation logic. Use Abstract Factory when you need to create families of related products or need consistent usage among products that need to be used together.

Abstract Factory Pattern - UML





Abstract Factory Pattern Example Code

```
public interface IvehicleFactory{
    ICar CreateCar();
    ILorry CreateLorry();
}

public class CityVehicleFactory : IvehicleFactory {
    public ICar CreateCar() {
        return new CityCar();
    }
    public ILorry CreateLorry(){
        return new CityLorry();
    }
}

public class OutOfTownVehicleFactory : IvehicleFactory {
    public ICar CreateCar() {
        return new OffroadCar();
    }
    public ILorry CreateLorry() {
        return new IntercityLorry();
    }
}
```

```
public interface Icar {
    void Drive();
}

public interface ILorry {
    void LoadCargo();
}

public class CityCar : Icar {
    public void Drive() {
        Console.WriteLine("A city car is on the move");
    }
}

public class CityLorry : ILorry {
    public void Loadcargo() {
        Console.WriteLine("City lorry cargo is loaded");
    }
}

public class OffroadCar {...}
Public class IntercityLorry {...}
```

```
IvehicleFactory cityFactory = new CityVehicleFactory();           Client Code
ICar citycar = cityFactory.CreateCar();
ILorry cityLorry = cityFactory.CreateLorry();
citycar.Drive();
cityLorry.LoadCargo();

// Create out of town vehicles
IvehicleFactory outoftownFactory = new OutOfTownVehicleFactory();
ICar offroadCar = outoftownFactory.CreateCar();
ILorry offroadLorry = outoftownFactory.CreateLorry();
offroadCar.Drive();
offroadLorry.Loadcargo();
```

DEMO: The Abstract Factory Pattern

The design effectively demonstrates the Abstract Factory pattern, facilitating easy expansion and modification while maintaining the separation of concerns between the creation logic and the usage of products.

Explanation of the Example

- The IVehicleFactory interface acts as the abstract factory, with methods for creating both cars and trucks.
- CityVehicleFactory and OffroadVehicleFactory are concrete factories that produce specific products (city or offroad vehicles).
- ICar and ITuck are abstract product interfaces, which have different concrete implementations based on the vehicle type (city or offroad).
- The client uses the abstract factory to create abstract products. This way, the client is decoupled from the creation of the actual products, and adding new vehicle types or environments would not require changes to the client code.

The Builder Pattern - Purpose

The Builder pattern is used to decouple the construction of a complex object from its representation

This pattern is particularly useful when an object must be created such that the same process can yield different results.

The Builder pattern is used to construct a complex object step by step, with the final step being the return the created object. The process of constructing an object should be generic so that it can be used to create different representations of the same object.

The Builder pattern is used to overcome three significant issues with factory and abstract factory design patterns in situations where the object contains a lot of attributes.

- Too many arguments to pass from the client program to the factory class can be error prone because most of the time, the type of arguments are the same and from the client side, it's hard to maintain the order of the argument.
- Some of the parameters might be optional, but in the factory pattern we are forced to send all the parameters and optional parameters need to send as null.
- If the object is heavy and its creation is complex, then all that complexity will be part of factory classes which can be confusing.

We can solve the issues with many parameters by providing a constructor with required parameters and then different setter methods to set the optional

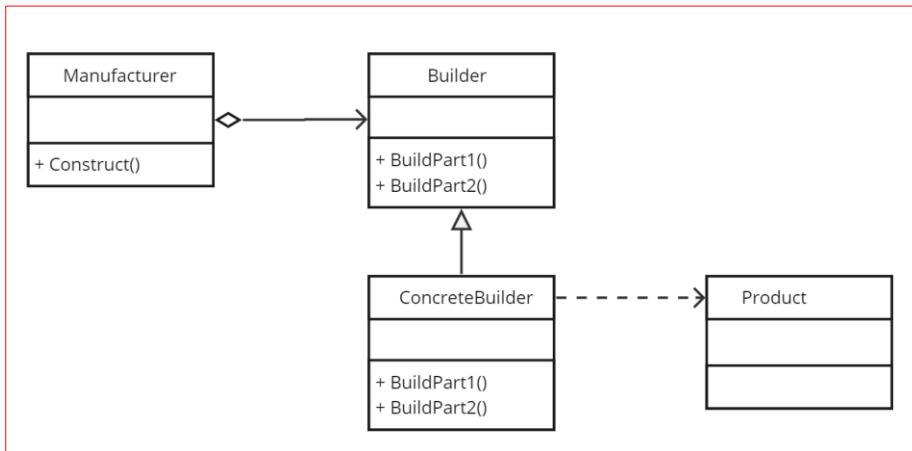
parameters. The problem with this approach is that the object state will be inconsistent unless all the attributes are set explicitly. The Builder pattern solves the issue with a large number of optional parameters and inconsistent state by providing a way to build the object step by step and provide a method that will actually return the final object.

The Builder Pattern - Implementation

Components of the Builder Pattern

- Builder: Specifies an abstract interface for creating parts of a Product object.
- ConcreteBuilder: Constructs and assembles parts of the product by implementing the Builder interface. It defines and keeps track of the representation it creates and provides an interface for retrieving the product.
- Director: Constructs an object using the Builder interface.
- Product: Represents the complex object being built

Builder Pattern - UML



The Builder Pattern Example Code - 1

Product Code

```
public class Car {  
    private List<string> _parts = new List<string>();  
    public Cartype Type { get; set; }  
  
    public void Add(string part) => _parts.Add(part);  
    public void Show(){  
        Console.WriteLine($"{this.Type} Car\n\tProduct Parts -----");  
        _parts.ForEach(part => Console.WriteLine($" \t{part}"));  
    }}}
```

Builder Code

```
public abstract class CarBuilder {  
    protected Cartype cartype { get; init; }  
    protected Car car;  
    public Car GetCar() => car;  
    public void CreateNewCar() =>  
        car = new Car() {Type = cartype};  
    public abstract void BuildChassis();  
    public abstract void BuildEngine();  
    public abstract void BuildWheels();  
    public abstract void BuildDoors();  
}
```

Product Type Code

```
public enum Cartype{  
    Saloon,  
    Estate,  
    SUV,  
    Sports  
}
```

The Builder Pattern Example Code - 2

Concrete Builder Code (Repeat for other car types)

```
public class EstateCarBuilder: CarBuilder {
    public EstateCarBuilder() => carType = CarType.Estate;

    public override void BuildChassis() => car.Add("Estate chassis");
    public override void BuildEngine() => car.Add("2000 cc");
    public override void BuildWheels() => car.Add("4 wheels");
    public override void BuildDoors() => car.Add("5 doors");
}
```

Manufacturer Code

```
public class Manufacturer {
    public void Construct(CarBuilder carBuilder) {
        carBuilder.CreateNewCar();
        carBuilder.BuildChassis();
        carBuilder.BuildEngine();
        carBuilder.BuildWheels();
        carBuilder.BuildDoors();
    }
}
```

Client Code

```
static void Main(string[] args) {
    List<Car> cars = new List<Car>();
    Manufacturer manufacturer = new Manufacturer();

    CarBuilder builder = new SaloonCarBuilder();
    manufacturer.Construct(builder);
    Car car = builder.GetCar();
    cars.Add(car);

    builder = new EstateCarBuilder();
    manufacturer.Construct(builder);
    car = builder.GetCar();
    cars.Add(car);

    cars.ForEach(c => c.Show());
}
```

DEMO: The Builder Pattern

The client (Program class) creates a Director object and a ConcreteBuilder object and passes the ConcreteBuilder to the Director. The director instructs the builder about what parts to build and in what order. After the building process, the builder returns the product to the client.

This approach allows different types of cars to be constructed using the same building process. It encapsulates the construction logic for a specific type of car within a ConcreteBuilder class, which can be replaced with a different ConcreteBuilder to construct a different type of car. This separation of concerns promotes reusability and flexibility.

Explanation of the Example

- **Car** (Product) represents the complex object under construction.
- **CarBuilder** (Builder) provides an abstract interface for creating parts of the Car.
- **EstateCarBuilder**, **SaloonCarBuilder**, **SportsCarBuilder** and **SUVCarBuilder** (ConcreteBuilders) construct and assemble parts of the product implementing the CarBuilder interface. Each has a different representation of the car.
- **Manufacturer** (Director) constructs an object using the CarBuilder interface.

The Prototype Pattern - Purpose

The Prototype pattern is used when the type of created objects is determined by a prototypical instance, and the objects are created by this prototype.

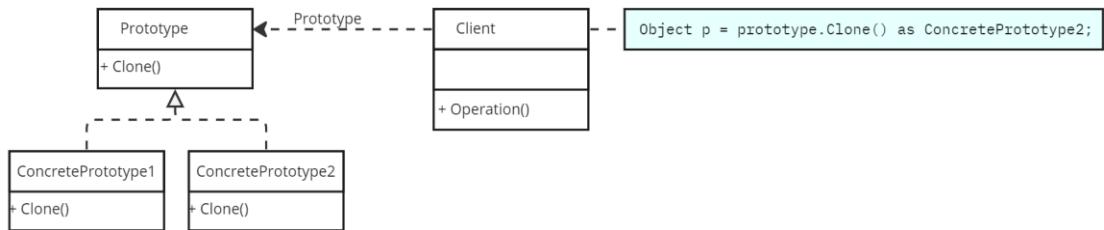
- Useful when the creation of an object involves costly or complex operations.
- Serve a copy of an existing prototype instance rather than creating one from scratch
- This pattern helps in:
- Reducing the need for creating subclasses.
- Hiding the complexities of making new instances from the client.
- Optimizing performance when the initialization of a class is resource-intensive.

The Prototype Pattern - Implementation

In C#, the Prototype pattern can be implemented using the **ICloneable** interface, which defines a method `Clone()` for cloning an object. It's important to know the difference between shallow and deep copying:

- **Shallow copy:** duplicates as little as possible. A shallow copy of an object is a new object whose instance variables are identical references to the corresponding instance variables in the original.
- **Deep copy:** duplicates everything. A deep copy of an object is a new object with entirely new instances of the duplicable elements of the object.

Prototype Pattern - UML



The Prototype Pattern Example Code

IPrototype Code

```
public interface IPrototype
{
    IPrototype Clone();
}
```

Client Code

```
ConcretePrototype prototype = new
ConcretePrototype("Sadia Saleem", 30);
ConcretePrototype clone = prototype.Clone() as
ConcretePrototype;

// Display original and cloned object
prototype.DisplayInfo();
clone.DisplayInfo();

// Making changes to verify that deep copy is indeed
clone.Name = "Isabelle Necessary";
clone.Age = 25;

Console.WriteLine("After changes to cloned object:");
prototype.DisplayInfo();
clone.DisplayInfo();
```

ConcretePrototype Code

```
public class ConcretePrototype : IPrototype {
    public int Age { get; set; }
    public string Name { get; set; }

    // constructor for initializing data
    public ConcretePrototype(string name, int age) {
        this.Name = name;
        this.Age = age;
    }

    // Copy Constructor for deep copying
    public ConcretePrototype(ConcretePrototype prototype) {
        Name = prototype.Name;
        Age = prototype.Age;
    }

    // Cloning method
    public IPrototype Clone() => new ConcretePrototype(this);
    public void DisplayInfo() =>
        Console.WriteLine($"Name: {Name}, Age: {Age}");
}
```

DEMO: The Prototype Pattern

The example demonstrates the flexibility of the Prototype pattern, enabling easy and efficient creation of new objects by copying existing instances. This can lead to performance improvements in systems where object creation is a bottleneck.

Explanation of the Example

- The IPrototype interface with a Clone() method for cloning objects.
- ConcretePrototype implements this interface and includes a copy constructor for creating a deep copy of itself.
- The Main() method creates an instance of ConcretePrototype, uses the Clone() method to create a deep copy, and changes the properties of the cloned object to demonstrate that the original object remains unaffected, proving the deep copy.

Structural Patterns

Simplify the design of large codebases by managing the relationships between entities.

These patterns help to organize the way classes and objects are composed

- They aim to :
- Simplify Complex Structures
- Promote Code Reusability
- Increase Flexibility
- Enhance Maintainability



Structural coding patterns are design patterns primarily focused on simplifying the design of large codebases by managing the relationships between entities. These patterns help to organize the way classes and objects are composed or combined, making the overall architecture more manageable and extendable.

Structural Coding patterns aim to:

- **Simplify Complex Structures:** By organizing the code into simpler, more understandable structures, these patterns help in reducing complexity and enhancing readability.
- **Promote Code Reusability:** Structural patterns often encourage the reuse of existing code, reducing redundancy and improving efficiency in the development process.
- **Increase Flexibility:** These patterns make the system easier to configure and adapt without altering the existing codebase significantly. This is particularly useful for adapting to new requirements or technologies.
- **Enhance Maintainability:** By defining clear relationships and interactions between different parts of the code, structural patterns improve maintainability, making it easier to update and modify code with fewer bugs and issues.

Structural Pattern Classifications

- Adapter
- Decorator
- Composite
- Facade



The Adapter Pattern - Purpose

Used when a client expects an object of a specific type and there's a legacy or third-party API offering the same functionality, but via an incompatible interface.

The adapter pattern helps to connect to the code by defining a different interface.

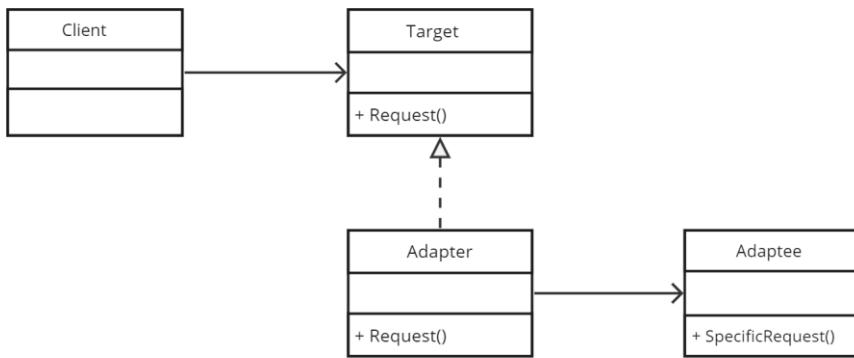
The adapter design pattern is one of the structural design patterns and its used so that two unrelated interfaces can work together. The object that joins this unrelated interface is called an adapter. This pattern is easy to understand as the real world is full of adapters. For example, consider a USB to Ethernet adapter. We need this when we have an Ethernet interface on one end and USB on the other. Since they are incompatible with each other, we use an adapter that converts one to another. This example is pretty analogous to object-oriented adapters. In design, adapters are used when we have a class, client, expecting some type of object, and we have an object, adaptee, offering the same features but exposing a different interface.

The Adapter Pattern - Implementation

There are two primary ways to implement the Adapter pattern one of which can't be used in C#::

- Object Adapter: Uses composition to contain the adaptee in the adapter object. This approach uses the adapter's instance variables to hold a reference to an instance of the adaptee.
- Class Adapter: Uses multiple inheritance to adapt one interface to another (**not applicable in C#** due to its lack of support for multiple inheritance at the class level).

Adapter Pattern - UML



The Adapter Pattern Example Code

ITarget Code

```
public interface ITarget {  
    void Request();  
}
```

Adapter Code

```
// The 'Adapter' class implements the ITarget  
// interface and wraps an instance of  
// the Adaptee class.  
  
public class Adapter: ITarget {  
    private Adaptee _adaptee;  
  
    public Adapter(Adaptee adaptee) =>  
        _adaptee = adaptee;  
  
    // Possibly doing some other work  
    // and then adapting  
    // the Adaptee's SpecificRequest to  
    // the Target's Request.  
    public void Request() =>  
        _adaptee.SpecificRequest();  
}
```

Adaptee Code

```
// The 'Adaptee' class contains some useful behavior,  
// but its interface is incompatible with the existing  
// client code. The Adaptee needs some adaptation  
// before the client code can use it.  
public class Adaptee {  
    public void SpecificRequest() =>  
        Console.WriteLine("Called SpecificRequest()");  
}
```

Client Code

```
public void Main() {  
    // Adaptee's methods used via the Target interface.  
    Adaptee adaptee = new Adaptee();  
    ITarget target = new Adapter(adaptee);  
    target.Request();  
}
```

DEMO: The Adapter Pattern

By using the Adapter pattern, existing classes with incompatible interfaces can be made to work together without modifying their source code, thus promoting reusability and flexibility within the system.

Explanation of the Example

- **ITarget** is the interface that the client uses.
- **Adaptee** is the class that has some specific functionality (**SpecificRequest**), but its interface is not what the client expects or can use.
- **Adapter** is the class that implements the ITarget interface, holds a reference to an Adaptee object, and translates ITarget requests into Adaptee specific calls. This translation is usually one-to-one but can involve some transformation of the data passed between the Adapter and the Adaptee.
- **Client** uses the Target interface (ITarget) to call the Adaptee's method through the Adapter. The client is completely decoupled from the implementation of the adaptee, only knowing about the ITarget interface.

The Decorator Pattern - Purpose

- Adds additional features or behaviours to an instance of a class while not modifying the other instances.
- Flexible alternative to subclassing for extending functionality.
- Solves problems such as
 - How can a class be reused if it doesn't have an interface that a client needs?
 - How can classes with incompatible interfaces work together?
 - How can a class be given an alternative interface?

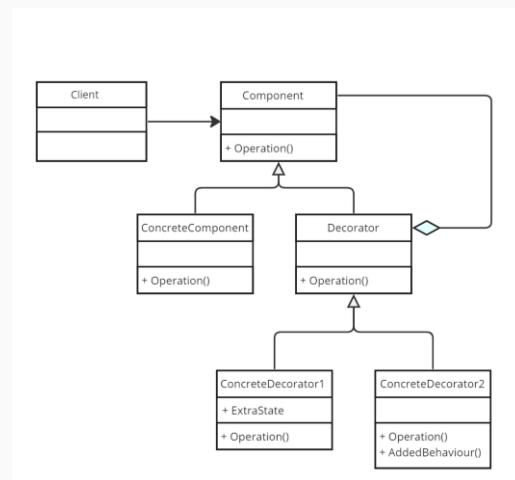
The decorator design pattern is used to add additional features or behaviours to a particular instance of a class while not modifying any other instances of the same class. Decorators provide a flexible alternative to subclassing for extending functionality. It is important to fully understand the decorator pattern because once you know the techniques of decorating, you'll be able to give your or someone else's objects new responsibilities without making any code changes to the underlying classes.

The Decorator Pattern - Implementation

The Decorator pattern typically involves the following roles:

- Component: An interface designed for objects to which responsibilities can be dynamically assigned..
- ConcreteComponent: Describes an object that can have extra responsibilities attached to it.
- Decorator: Holds a reference to a Component object and provides an interface that matches the interface of the Component.
- ConcreteDecorator: Gives the component additional responsibilities.

Decorator Pattern - UML





The Decorator Pattern Example Code

Component Code

```
public abstract class Beverage {  
    public abstract string Description { get; }  
    public abstract decimal Cost { get; }  
}
```

ConcreteComponent Code

```
public class Espresso : Beverage {  
    public override string Description { get; } = "Espresso";  
    public override decimal Cost { get; } = 2.99m;  
}
```

Decorator Code

```
public abstract class CondimentDecorator : Beverage  
{  
    public abstract override string Description { get; }  
}
```

ConcreteDecorator Code

```
public class SteamedMilk : CondimentDecorator {  
    private Beverage _bev;  
    public SteamedMilk(Beverage bev) => _beverage = bev;  
  
    public override string Description {  
        get => _bev.Description + ", Steamed Milk";  
    }  
  
    public override decimal Cost {  
        get => _bev.Cost + 0.50m;  
    }  
}
```

Client Code

```
public void Main() {  
    List<Beverage> beverages = new List<Beverage>();  
  
    Beverage beverage;  
    beverage = new Espresso();  
    beverages.Add(beverage);  
  
    beverage = new Espresso();  
    beverage = new SteamedMilk(beverage);  
    beverages.Add(beverage);  
  
    beverages.ForEach(beverage =>  
        Console.WriteLine($"{beverage.Description}  
        {beverage.Cost:$0.00}");  
}
```

DEMO: The Decorator Pattern

This example of a coffee ordering system demonstrates how you can dynamically add functionality to an object without altering its structure. When the functionality of a component needs to be extended, the decorator pattern can be used as a flexible alternative to subclassing.

Explanation of the Example

- **Beverage** (Component) defines the interface for objects that can have responsibilities added to them dynamically.
- **Espresso** (ConcreteComponent) defines an object (Espresso) to which additional responsibilities can be attached.
- **CondimentDecorator** (Decorator) maintains a reference to a Beverage object and defines an interface that conforms to Beverage's interface.
- **ChocolateSauce** and **SteamedMilk** (ConcreteDecorators) add responsibilities to the Beverage objects they decorate. They modify the behaviour of these objects by adding their own behaviour before or after delegating the call to the object they decorate.

The Composite Pattern - Purpose

The Composite pattern allows you to compose objects into tree-like structures to represent hierarchies where a group of objects is treated in the same way as a single instance of the same object type.

- This pattern is particularly useful when:
- You need to implement a hierarchical structure .
- You want clients to ignore the differences between compositions of objects and individual objects.

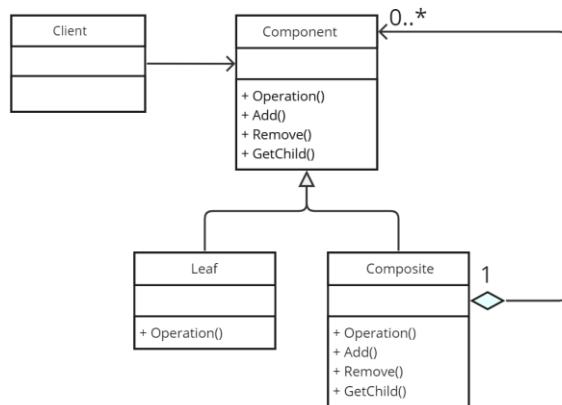
The Composite pattern allows you to compose objects into tree-like structures to represent hierarchies where a group of objects is treated in the same way as a single instance of the same object type. This pattern creates a uniform interface for individual objects and groups of objects enabling clients to treat individual objects and compositions of objects uniformly.

The Composite Pattern - Implementation

The composite pattern involves the following:

- **Component:**
 - This is an abstract class or interface with common operations for managing children.
 - It usually includes methods like Add, Remove, and GetChild along with business logic functionality.
- **Leaf:**
 - Represents end objects in a composition.
 - A leaf can't have any children.
- **Composite:**
 - Defines behaviour for components having children.
 - Stores child components and implements child-related operations in the Component interface.

Composite Pattern - UML





The Composite Pattern Example Code

Component Code

```
public abstract class Component {  
    protected string name;  
    public Component(string name) => this.name = name;  
    public abstract void Add(Component component);  
    public abstract void Remove(Component component);  
    public abstract void Display(int depth);  
}
```

Composite Code

```
public class Composite : Component {  
    private List<Component> _children =  
        new List<Component>();  
  
    public Composite(string name) : base(name) {}  
  
    public override void Add(Component component) =>  
        _children.Add(component);  
  
    public override void Remove(Component component) =>  
        _children.Remove(component);  
  
    public override void Display(int depth) {  
        Console.WriteLine(new String('-', depth) + name);  
        _children.ForEach(comp =>  
            comp.Display(depth + 2));  
    }  
}
```

Leaf Code

```
public class Leaf : Component {  
    public Leaf(string name) : base(name) {}  
    public override void Add(Component component) =>  
        Console.WriteLine("Cannot add to a leaf");  
  
    public override void Remove(Component component) =>  
        Console.WriteLine("Cannot remove from a leaf");  
  
    public override void Display(int depth) =>  
        Console.WriteLine(new String('-', depth) + name);  
}
```

Client Code

```
Composite root = new Composite("root");  
root.Add(new Leaf("Leaf A"));  
root.Add(new Leaf("Leaf B"));  
  
Composite comp = new Composite("Composite X");  
comp.Add(new Leaf("Leaf XA"));  
comp.Add(new Leaf("Leaf XB"));  
  
root.Add(comp);  
root.Add(new Leaf("Leaf C"));  
  
Leaf leaf = new Leaf("Leaf D");  
root.Add(leaf);  
root.Remove(leaf);  
  
root.Display(1);
```

DEMO: The Composite Pattern

This structure allows you to work through the entire tree using a single interface, Component. Any operation like Display can be performed on either a single leaf or a complex composition of nodes, treating them uniformly. This encapsulation allows flexibility and ease of maintenance.

Explanation of the Example

- The **Component** abstract class provides a common interface for all objects in the composition, both simple and complex.
- **Leaf** represents end objects in the structure. Leaf objects cannot have any children, so Add and Remove methods in Leaf class simply print a message indicating that these operations are not possible.
- **Composite** is the class that can have children (leaf nodes or other composites). It implements Add, Remove, and Display methods. The Display method prints an object and its children, recursively showing the structure.
- In the **Main** function, we create a tree structure with one root node (root), to which we add leaf nodes and a composite node (comp). We then manipulate the tree by adding and removing nodes, and finally, we call Display to show the structure.

The FAÇADE Pattern - Purpose

The Facade pattern provides a simplified interface to a more complicated subsystem.

- Does **NOT** encapsulate the subsystem.
- It provides a higher-level interface that makes the subsystem easier to use.
 - This is particularly useful when working with a large body of code, such as a complex library or API, where the direct management of objects within the subsystem can be complex or cumbersome.
- The facade simplifies the complexity behind the system, minimizing the communication and dependencies between systems.
 - Promotes loose coupling.
- Clients need only interact with the facade instead of the complex underlying system.

The FAÇADE Pattern - Implementation

The Façade pattern involves the following:

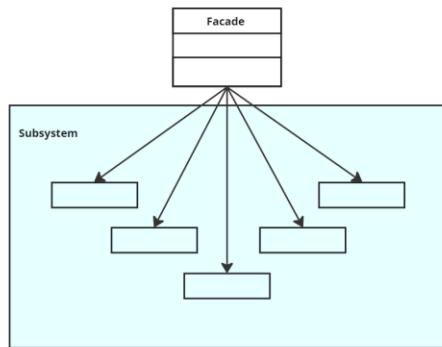
- **Facade:**

- Provides a simple interface to the complex logic of one or several subsystems.
- It delegates client requests to appropriate objects within the subsystem and manages their lifecycle.

- **Subsystem Classes:**

- Implement system functionality and perform the actual work.
- These classes are often fine-grained and intricate, requiring detailed knowledge to use correctly.

Facade Pattern - UML



The Facade Pattern Example Code – Part 1

DVD Code

```
// Subsystem Class 1
public class DvdPlayer {
    public void On() =>
        Console.WriteLine("DVD Player is on.");
    public void Play(string film) =>
        Console.WriteLine($"Playing \'{film}\'.");
    public void Off() =>
        Console.WriteLine("DVD Player is off.");
}
```

Projector Code

```
// Subsystem Class 2
public class Projector {
    public void On() =>
        Console.WriteLine("Projector is on.");
    public void SetInputDvd(DvdPlayer dvd) =>
        Console.WriteLine("Projector i/p set to DVD Player.");
    public void WidescreenMode() =>
        Console.WriteLine("Projector set to widescreen mode.");
    public void Off() =>
        Console.WriteLine("Projector is off.");
}
```

TheatreLights Code

```
// Subsystem Class 3
public class TheatreLights {
    public void Dim(int level) => Console.WriteLine(
        $"Theatre lights dimmed to {level}%.");
    public void On() =>
        Console.WriteLine("Theatre lights are on.");
}
```

The Facade Pattern Example Code – Part 2

Facade Code

```
public class HomeTheatreFacade {  
    private DvdPlayer _dvd;  
    private Projector _projector;  
    private TheatreLights _lights;  
  
    public HomeTheatreFacade(DvdPlayer dvd, Projector projector, TheatreLights lights) {  
        _dvd = dvd;  
        _projector = projector;  
        _lights = lights;  
    }  
  
    public void WatchFilm(string film) {  
        Console.WriteLine("Get ready to watch a Film...");  
        _lights.Dim(10);  
        _projector.On();  
        _projector.setInputDvd(_dvd);  
        _projector.WideScreenMode();  
        _dvd.On();  
        _dvd.Play(film);  
    }  
  
    public void EndFilm() {  
        Console.WriteLine("Shutting down the Film theatre...");  
        _dvd.Off();  
        _projector.Off();  
        _lights.On();  
    }  
}
```

The Facade Pattern Example Code – Part 3

Client Code

```
static void Main(string[] args) {
    DvdPlayer dvd = new DvdPlayer();
    Projector projector = new Projector();
    TheatreLights lights = new TheatreLights();

    HomeTheatreFacade homeTheater = new HomeTheatreFacade(dvd, projector, lights);
    homeTheater.WatchFilm("Postman Pat in 'The Heist!'");
    homeTheater.EndFilm();
}
```

DEMO: The Facade Pattern

Clients interact with the facade instead of the subsystem directly, which makes the client code cleaner, easier to maintain, and less coupled to the inner workings of the subsystems. This setup embodies the essence of the Facade pattern by hiding the complexities of the subsystem and providing a simpler interface to the client.

Explanation of the Example

- **Subsystem** classes (**DvdPlayer**, **Projector**, and **TheatreLights**) perform specific tasks and represent various parts of the home theatre system. Each class has a detailed interface that can be complex to manage directly.
- **HomeTheatreFacade** provides a simplified interface (**WatchMovie** and **EndMovie** methods) to the more complex parts of the home theatre system. It handles all the details of setting up for movie viewing and shutting down afterwards, which simplifies the client's responsibilities.

Behavioural Patterns

Focus on improving communication and responsibility distribution among objects

- Manage Complex Control Flows
- Encapsulate Requests
- Enhance Flexibility in Interaction
- Improve Scalability and Maintainability
- Promote Loose Coupling



Behavioural Coding patterns are used to:

1. Manage Complex Control Flows: Behavioral patterns simplify complex control flows by managing and distributing responsibilities between various objects. This often involves defining a clear protocol or chain of responsibility which objects follow to handle a particular task.

2. Encapsulate Requests: These patterns can encapsulate information needed to perform actions or trigger events. This encapsulation allows for greater flexibility in specifying, queuing, and executing requests at different times.

3. Enhance Flexibility in Interaction: By delegating how and when certain interactions occur between objects, behavioral patterns increase the flexibility and efficiency of communication.

4. Improve Scalability and Maintainability: Behavioral patterns can help make a system easier to scale and maintain by localizing changes to the behavior in a few objects or through standardized interfaces, without necessitating widespread modifications throughout the codebase.

5. Promote Loose Coupling: Many behavioral patterns aim to reduce dependencies between objects, which promotes loose coupling and fewer direct relationships. This makes systems easier to modify and extend.

Behavioural Pattern Classifications

- Strategy
- Observer
- Command



The Strategy Pattern - Purpose

The Strategy pattern allows algorithms to be dynamically selected at runtime.

The main idea behind the Strategy pattern is to:

- Define a family of algorithms
- Encapsulate each one as an object
- Make them interchangeable.
- Allow the chosen algorithm to vary independent of the clients that use it.

The Strategy pattern is typically used to:

- Manage algorithms, relationships, and responsibilities between objects.
- Decouple the class operations from the behaviour that may vary often.
- Provide alternative ways of executing the same task.

The Strategy pattern allows algorithms to be dynamically selected at runtime.

Rather than implement a single algorithm directly, the code receives instructions at run-time that specify which from a set of algorithms to use.

The Strategy Pattern - Implementation

The Strategy pattern uses the following:

- **Context:**

- Maintains a reference to one of the concrete strategies.
- Communicates with this object only via the strategy interface.

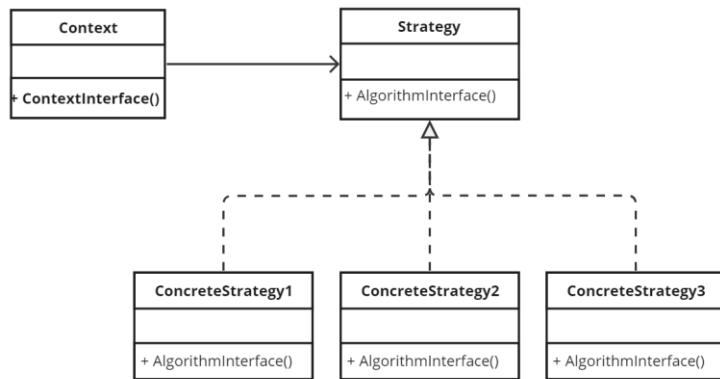
- **Strategy:**

- Defines an interface common to all supported algorithms.
- Context uses this interface to call the algorithm defined by a ConcreteStrategy.

- **ConcreteStrategy:**

- Implements the algorithm using the Strategy interface.

Strategy Pattern - UML



The Strategy Pattern Example Code – Part 1

Context Code

```
public class Context {  
    private IStrategy _strategy;  
    public Context(IStrategy strategy) =>  
        _strategy = strategy;  
    public void SetStrategy(IStrategy strategy) =>  
        _strategy = strategy;  
    public void ExecuteStrategy() =>  
        _strategy.Execute();  
}
```

IStrategy Code

```
public interface IPaymentStrategy {  
    void ProcessPayment(double amount);  
}
```

ConcreteStrategy1 Code

```
public class CreditCardPayment : IPaymentStrategy {  
    private string _cardNumber;  
    private string _cvv;  
    private string _expiryDate;  
  
    public CreditCardPayment(  
        string cardNumber, string cvv, string expiryDate){  
        _cardNumber = cardNumber;  
        _cvv = cvv;  
        _expiryDate = expiryDate;  
    }  
  
    public void ProcessPayment(double amount) {  
        Console.WriteLine(  
            $"Processing credit card payment for {amount:C}");  
        // Implementation for processing credit card payment  
    }  
}
```

The Strategy Pattern Example Code – Part 2

ConcreteStrategy2 Code

```
public class PayPalPayment : IPaymentStrategy
{
    private string _emailAddress;
    private string _password;

    public PayPalPayment(string email, string password) {
        _emailAddress = email;
        _password = password;
    }

    public void ProcessPayment(double amount) {
        Console.WriteLine(
            $"Processing PayPal payment for {amount:c}");
        // Implementation for processing PayPal payment
    }
}
```

Client Code

```
static void Main(string[] args) {
    var paymentContext = new PaymentContext();

    var creditCardPayment =
        new CreditCardPayment("1234567890123456",
            "123", "12/25");
    paymentContext.SetPaymentStrategy(
        creditCardPayment);
    paymentContext.Pay(125.75);

    var paypalPayment =
        new PayPalPayment("user@example.com",
            "password123");
    paymentContext.SetPaymentStrategy(
        paypalPayment);
    paymentContext.Pay(89.50)
}
```

DEMO: The Strategy Pattern

This approach demonstrates the power of the Strategy pattern to facilitate adding new payment methods without changing the existing codebase significantly. It enhances the system's flexibility and makes it easier to extend. Each payment method can be developed, tested, and maintained independently, promoting better code organization and separation of concerns.

Explanation of the Example

- **IPaymentStrategy:** Defines a common interface for all supported payment algorithms.
- **CreditCardPayment** and **PayPalPayment:** Implement the payment process for specific payment methods.
- **PaymentContext:** Uses a payment strategy to delegate the payment processing to the strategy object. It allows for changing the payment strategy dynamically depending on the user's choice.

The Observer Pattern - Purpose

The Observer pattern specifies one-to-many dependencies between objects:

- When an object's state changes, all its dependents are notified and updated automatically.

The pattern is useful in the following scenarios:

- When a change to one object requires changing others, and you don't necessarily know how many objects need to be changed.
- It helps to establish a subscription mechanism to send notifications to multiple objects about any events that happen to the object they're observing.

The Observer pattern specifies one-to-many dependencies between objects:

- When an object's state changes, all its dependents are notified and updated automatically.
- The object that maintains the list of dependents (observers) is generally called the "subject" and its dependents are known as the "observers".

The pattern is useful in the following scenarios:

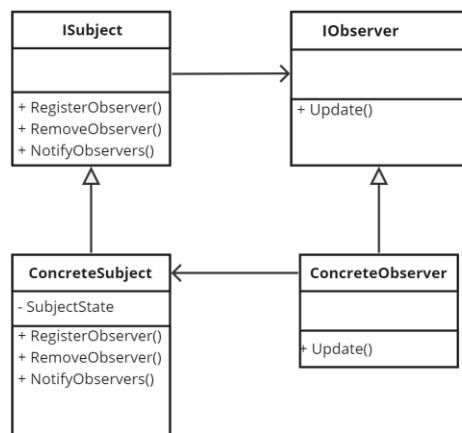
- When a change to one object requires changing others, and you don't necessarily know how many objects need to be changed.
- It helps to establish a subscription mechanism to send notifications to multiple objects about any events that happen to the object they're observing.

The Observer Pattern - Implementation

The Observer pattern uses the following:

- **Subject**
 - Manages a list of observers
 - Facilitates the addition and removal of observers.
- **Observer**
 - Provides an update interface for interested parties that need to be informed about changes to a subject.
- **Concrete Subject**
 - Stores relevant state for the ConcreteObserver.
 - When its state changes. It sends a notification to its observer(s).
- **Concrete Observer**
 - Maintains a reference to a ConcreteSubject object
 - Stores state that remains consistent with the state of the subject by implementing the Observer's updating interface.

Observer Pattern - UML



The Observer Pattern Example Code – Part 1

Subject Code

```
public interface Isubject {  
    void RegisterObserver(Iobserver observer);  
    void RemoveObserver(Iobserver observer);  
    void NotifyObservers();  
}
```

ConcreteSubject Code

```
public class WeatherDataSubject : Isubject {  
    private List<Iobserver> observers;  
    private float temperature;  
    private float humidity;  
    private float pressure;  
  
    public WeatherDataSubject() =>  
        observers = new List<Iobserver>();  
    public void RegisterObserver(Iobserver observer) =>  
        observers.Add(observer);  
    public void RemoveObserver(Iobserver observer) =>  
        observers.Remove(observer);  
    public void NotifyObservers() {  
        observers.ForEach(o =>  
            o.Update(temperature, humidity, pressure));  
    }  
  
    public void MeasurementsChanged() => NotifyObservers();  
    public void SetMeasurements(float temp, float hum, float pres) {  
        this.temperature = temp;  
        this.humidity = hum;  
        this.pressure = pres;  
        MeasurementsChanged();  
    }  
}
```



The Observer Pattern Example Code – Part 2

Observer Code

```
public interface Iobserver {  
    void Update(float temperature,  
               float humidity,  
               float pressure);  
}
```

Client Code

```
static void Main(string[] args) {  
    WeatherDataSubject wData =  
        new WeatherDataSubject();  
    CurrentConditionsObserver curDisplay = new  
        CurrentConditionsObserver(wData);  
  
    wData.SetMeasurements(27, 65, 1026f);  
    wData.SetMeasurements(28, 70, 1031f);  
}
```

ConcreteObserver Code

```
public class CurrentConditionsObserver : Iobserver {  
    private float temperature;  
    private float humidity;  
    private float pressure;  
    private ISubject weatherData;  
  
    public CurrentConditionsObserver(ISubject weatherData) {  
        this.weatherData = weatherData;  
        weatherData.RegisterObserver(this);  
    }  
  
    public void Update(  
        float temperature, float humidity, float pressure) {  
        this.temperature = temperature;  
        this.humidity = humidity;  
        this.pressure = pressure;  
        Display();  
    }  
  
    public void Display() {  
        Console.WriteLine(  
            $"Current conditions: {temperature}C degrees, " +  
            $"{humidity}% humidity and {pressure}hPa pressure");  
    }  
}
```

Observer Pattern Vs Delegates

- In .NET and C# you would normally use delegates and events to achieve the observer functionality
- Delegates:
 - Uses delegate and event to notify subscribers.
 - Subscribers simply attach methods to the event, making it less verbose.
 - Provides a more straightforward and idiomatic approach in C# for simpler use cases.
- Both alternatives achieve the same functionality: when the State property changes, all registered observers/subscribers are notified.

Delegate Example

Client Code

```
static void Main(string[] args) {
    var subject = new Subject();

    // Attach observers using delegates
    subject.StateChanged += (newState) =>
    {
        Console.WriteLine(
            $"Observer A update: State is now {newState}");
    };

    subject.StateChanged += (newState) =>
    {
        Console.WriteLine(
            $"Observer B update: State is now {newState}");
    };

    subject.State = 10; // Notifies all subscribers
    subject.State = 20; // Notifies all subscribers
}
```

ConcreteSubject Code

```
public class Subject
{
    public delegate void StatechangedHandler(int newState);
    public event StatechangedHandler StateChanged;

    private int state;
    public int State
    {
        get => state;
        set
        {
            state = value;
            OnStateChanged(state);
        }
    }

    protected virtual void OnStateChanged(int newState)
    {
        Statechanged?.Invoke(newState); // Notify all subscribers
    }
}
```

DEMO: The Observer Pattern

The example effectively demonstrates how the Observer pattern can facilitate the real-time update of multiple displays based on a single data source, maintaining consistency across different components of the system. This pattern is also common in various other domains like GUI toolkits, event management systems, and more complex business applications where actions are triggered by changes in state or status.

Explanation of the Example

- **WeatherData** is the Subject that maintains a list of observers (display devices) and notifies them of changes.
- **CurrentConditionsDisplay** is an Observer that updates itself whenever **WeatherData** changes.
- When **SetMeasurements** is called, it simulates new data coming from the weather station, which then calls **MeasurementsChanged**, triggering an update to all registered displays via **NotifyObservers**.

The Command Pattern - Purpose

The Command pattern turns a request into a standalone object that encapsulates all the information about the request that is required to perform a later action or trigger. Such as:

- Method name
 - The object that owns the method
 - Values for the method's parameters
- This transformation lets you:
- Pass requests as method arguments
 - Delay or queue request execution
 - Manage undoable operations.
 - Command pattern is a data-driven design pattern.



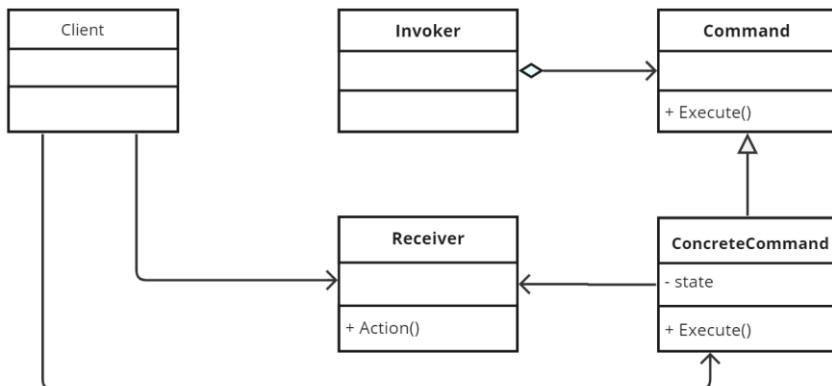
Essentially, the Command pattern encapsulates a command request as an object, thereby letting you parameterize clients with queues, requests, and operations.

The Command Pattern - Implementation

- **Command**
 - Specifies an interface for invoking an operation.
- **Concrete Command**
 - Extends the Command interface, implementing the invoking method by executing the Receiver's corresponding operation(s).
- **Receiver**
 - Knows how to carry out operations that are associated with a request. Any class can be a Receiver.
- **Invoker**
 - Triggers the command to deal with the request..
- **Client**
 - Specifies a ConcreteCommand object and sets its receiver.

A request is wrapped under an object as command and passed to invoker object. Invoker object looks for the appropriate object which can handle this command and passes the command to the corresponding object which executes the command. It's also known as action or transaction.

Command Pattern - UML



The command object knows about the receiver and invokes methods of the receiver by supplying parameters. Values for parameters of the receiver's methods are stored in the command object. The receiver objects perform the job when the execute() method of command gets called. We pass this command object to an invoker object to perform the execute() method. Invoker object executes the method of the command object and passes the required parameters to it.

The Command Pattern Example Code – Part 1

ICommand Code

```
public interface ICommand {  
    void Execute();  
}
```

ConcreteCommand1 Code

```
public class LightOffCommand : ICommand {  
    private Light _light;  
    public LightOffCommand(Light light) => _light = light;  
    public void Execute() => _light.Turnoff();  
}
```

ConcreteCommand2 Code

```
public class LightOnCommand : ICommand {  
    private Light _light;  
    public LightOnCommand(Light light) => _light = light;  
    public void Execute() => _light.Turnon();  
}
```

Receiver Code

```
public class Light {  
    public LightStatus Status { get; set; } = LightStatus.Off;  
    public void TurnOn() {  
        if (Status == LightStatus.Off) {  
            Status = LightStatus.On;  
            Console.WriteLine("The light is on");  
        }  
    }  
  
    public void TurnOff() {  
        if (Status == LightStatus.On) {  
            Status = LightStatus.Off;  
            Console.WriteLine("The light is off");  
        }  
    }  
}
```

The Command Pattern Example Code – Part 2

Invoker Code

```
public class RemoteControl {  
    private ICommand _onCommand;  
    private ICommand _offCommand;  
  
    public RemoteControl(ICommand onCommand, ICommand offCommand) {  
        _onCommand = onCommand;  
        _offCommand = offCommand;  
    }  
  
    public void PressOnButton() => _onCommand.Execute();  
    public void PressOffButton() => _offCommand.Execute();  
}
```

Client Code

```
static void Main(string[] args) {  
    Light light = new Light();  
    ICommand lightOn = new LightOnCommand(light);  
    ICommand lightOff = new LightOffCommand(light);  
  
    RemoteControl control = new RemoteControl(lightOn, lightOff);  
    control.PressOnButton(); // Output: The light is on  
    control.PressOffButton(); // Output: The light is off  
    control.PressOffButton(); // Output: NO OUTPUT  
    control.PressOnButton(); // Output: The light is on  
    control.PressOnButton(); // Output: NO OUTPUT  
}
```

Command Pattern Vs Func Delegate

- In .NET and C# you would normally use the Func delegate to achieve command pattern functionality
- Func Delegate:
 - Uses a lightweight approach where each operation is represented by a Func<string> delegate.
 - Operations are directly defined as lambda expressions or methods and executed in sequence.
 - Best suited for simpler scenarios where full encapsulation and decoupling are not required.
- Both alternatives achieve the same functionality.
 - The Command pattern provides a more structured and extensible approach,
 - The Func delegate offers simplicity and conciseness.

Func Delegate Example

Client Code

```
// Define a list of operations using Func
List<Func<string>> operations = new List<Func<string>>();

// Add operations
operations.Add(() =>
{
    int a = 5, b = 3;
    return $"Result of addition: {a + b}";
});

operations.Add(() =>
{
    int a = 5, b = 3;
    return $"Result of multiplication: {a * b}";
});

// Execute all operations
foreach (var operation in operations)
{
    Console.WriteLine(operation());
}
```



DEMO: The Command Pattern

This example of switching a light on and off allows the invoker (a remote control) to be decoupled from the object handling the invocation (light switches), adding flexibility and extensibility to the framework. The Command pattern is very useful when you need to support undo operations, track a history of commands, or manage transactions and rollbacks.

Explanation of the Example

- **Light** is the Receiver class. It knows how to perform the actual operations associated with turning a light on and off.
- **ICommand** is the Command interface with an Execute() method.
- **LightOnCommand** and **LightOffCommand** are ConcreteCommand classes that implement the ICommand interface and invoke operations on the Light.
- **RemoteControl** is the Invoker; it has a command object for each action (turning the light on and off) and it executes the command by calling the command's Execute() method when its button is pressed.
- **Program** class in Main() function acts as the Client, setting up the object relationships and starting commands.

Hands on Lab

Controllers and Actions

Objective:

To create a basic vehicle management system that uses a number of design patterns.

You are tasked with designing and implementing a vehicle management system in C#. This system will allow users to create, manage, and monitor different types of vehicles such as cars, lorries, and motorcycles. To ensure the application is scalable, maintainable, and well-organized, you will employ the Factory, Composite, and Observer design patterns.

Summary

Know more about the configuration and use of Controllers and Actions

- Introduction to Design Patterns
- Pattern Classifications
- Creational Patterns
 - Singleton, Factory, Builder, Prototype
- Structural Patterns
 - Adapter, Decorator, Composite, Facade
- Behavioural Patterns
 - Strategy, Observer, Command

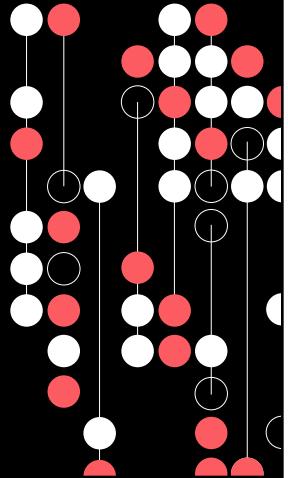


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Asynchronous Programming and Concurrency



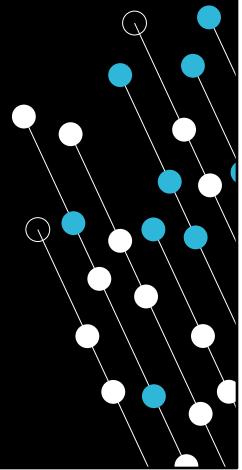
Session Objectives

- Grasp the concepts of asynchronous programming in C#
- Learn how to effectively use Tasks and Task Parallel Library (TPL) for asynchronous operations
- Know how to manage concurrency and understand best practices to avoid race conditions and deadlocks

Session Content

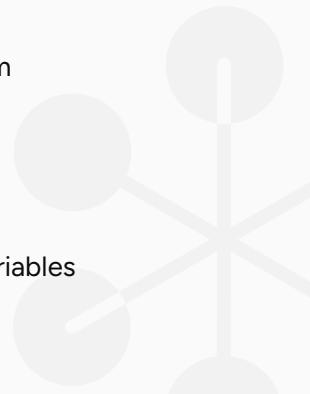
- Asynchronous Programming Patterns
- Task-based Asynchronous Patterns (TAP)
- Threading
- Task Parallel Library (TPL)

Using Threads



Threading - An Introduction

- How is your code executed?
 - Linearly
 - By a thread, which is scheduled by the operating system
- A thread needs to know
 - The instruction to start from
 - Which instruction to execute next
- A thread needs to have
 - A stack onto which it will place parameters and local variables
 - A place to copy the registers from the processor



Problems of a Single Threaded Application

- Freezing the user interface
 - Long tasks prevent messages from being pumped
 - Not so relevant for web UI's
- Network calls can take a long time to complete
 - Web Services or remoting
- Server can't afford to serve clients one after the other
 - Network protocols will time out
- Consequently, you might need to use multiple threads
 - Not a "silver bullet"
 - Massively increased code complexity

If you are single threaded, then while the thread is off executing some code, the rest of the app is starved of any processor time.

Important to stress that adding more than one thread will not magically make the application quicker, and the extra code complexity will make even experienced programmers wince.

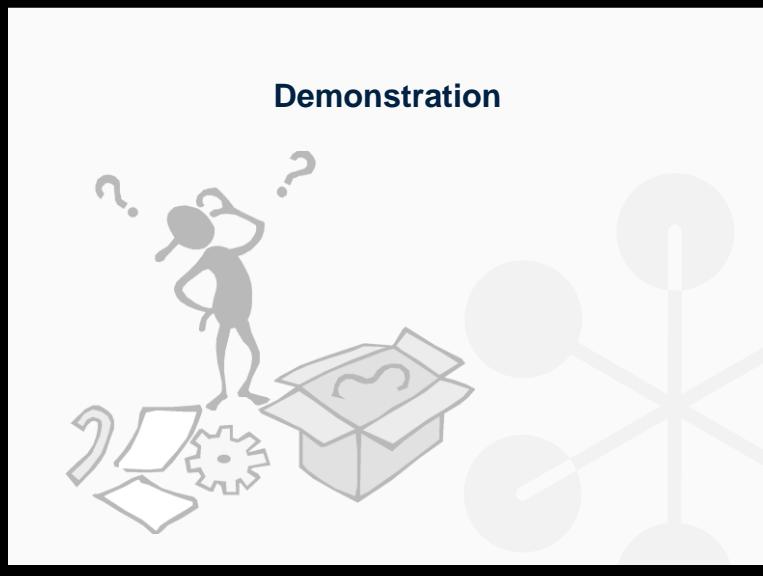
Managed Threads

- System.Threading namespace
 - System.Threading.Thread class
- Threads are scheduled by the operating system
- Threads are constrained to a process
 - But are shared across AppDomains
- Threads need an entry point
 - Specified using a ThreadStart delegate
- Threads have a priority
- Foreground threads can keep an AppDomain alive
 - Background threads will not



Basic overview of the System.Threading.Thread class. You might want to zip off to the MSDN help to explain some of this.

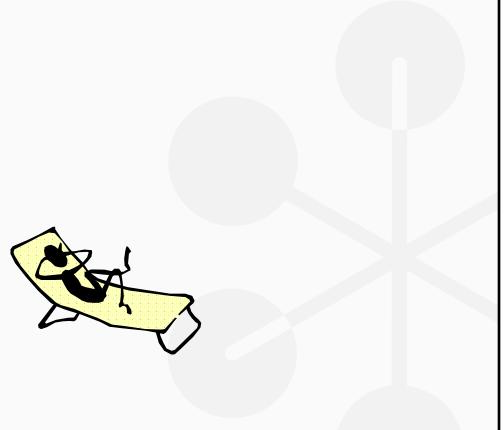
Creating Threads



Perhaps show how easy it is to create and start a thread via "ThreadingIsEasy" demo. Then show OutOfControl demo and demonstrate InControl (via Mutex lock)

Controlling Threads

- Start()
 - To start a thread
- Join()
 - To wait for another thread to finish
- Interrupt()
 - To wake a sleeping or blocked thread
- Sleep()
 - Called by a thread to send *itself* to sleep



Note that Sleep() is a method that a thread calls on itself, whereas the others are called on other threads.

C# used to support Abort, Suspend and Resume methods but these have all been discontinued and are obsolete.

Note that Interrupt() will throw an exception on the target thread.

Problems with Multi-Threading

- Lifetime management
 - Worker thread might complete before Start() returns
- Coordination of thread activities
 - Worker thread might process information too soon
 - Master thread needs to know when worker thread has finished
- Synchronisation
 - Simultaneous access to shared resources
 - Especially streams, object members



This might come as a major shock but there are huge problems and difficulties associated with the creation of multi-threaded programs, so be warned. Simple things like adding one to a number, or accessing a file, become problematic when there is more than one thread.

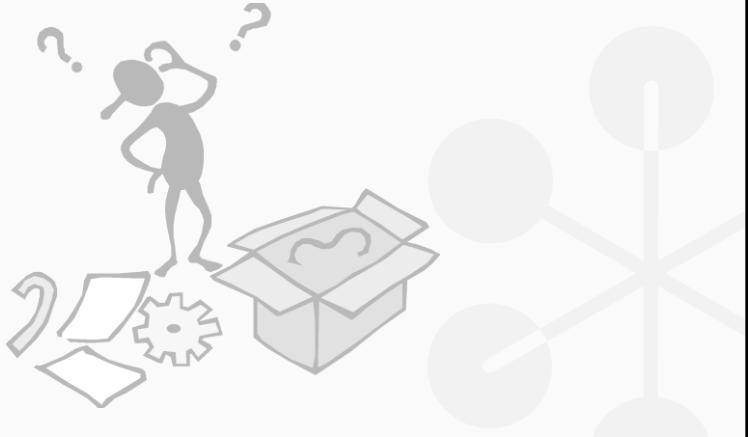
Coordinating Threads

- Join() and Interrupt()
 - Crude, but effective
- ManualResetEvent
 - Signal to a thread that it can proceed
 - Multiple threads can be released at the same time
- AutoResetEvent
 - Only a single thread is released at a time
- Both event classes extend the WaitHandle
 - WaitAll(), WaitAny(), WaitOne()
- Use Set() and Reset() to signal / un-signal the event



Coordinating Threads

Demonstration



Look at the ThreadRacer program, that shows how to use a ManualResetEvent to coordinate the actions of multiple threads. You can also look at ThreadTrain that looks at using locks.

Synchronisation

- Interlocked class
 - For simple numeric manipulation
- Monitor class
 - For controlling access to critical sections of code
- SyncLock and lock()
 - Easier use of the Monitor class
- Mutex class
 - Mutually exclusive access to critical resources
 - Can be used across AppDomains and processes

Talk through why synchronisation is needed, and explain the different classes. Stress that it is class members that are most at risk, as well as resources such as files, that will need protecting.

Interlocked class is for simple numerical control, as even adding one to a field in a class is not thread safe without it. Monitor is a code blocking mechanism, and you can use it either on an object, for shared members, or if you want synchronisation across all instances of a class, use the Type of the class as the target.

SyncLock in VB, and lock() in C# wrap up Monitor with exception handling, so that is not too hard.

Mutex offers cross-process support using named mutexes, which is useful if multiple processes might be writing to a single file, and you don't want to employ file share locking.

Synchronising Threads

Demonstration



Look at the ThreadTrain demo to see a demonstration of a mutex. Start multiple instances of the program to prove that it works across process boundaries.

Synchronisation Pitfalls

- Race conditions
 - Occurs when threads act in an uncoordinated manner
- Deadlock

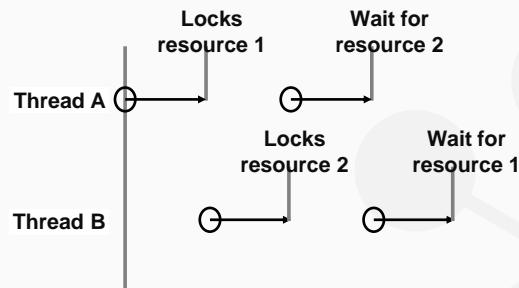


Figure 4-7 : Deadlock

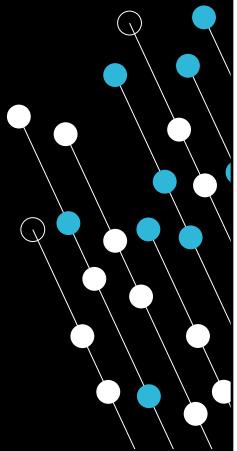
Think about the problems that might occur - in particular race conditions and deadlock. Note it is very hard to debug these, because any action that you take will affect how the threads run.

Alternatives to Creating Threads

- ThreadPool
 - Ready to run set of threads
 - Less flexibility than using your own thread
 - Easier to program
- System.Threading.Timer
 - Background timer that uses the ThreadPool
- Asynchronous Delegates
 - Use a ThreadPool thread to call a method

Just talk thru the alternatives. ThreadPool is easier to use, but less flexible. System.Threading.Timer (not to be confused with the graphical Timer from System.Windows.Forms) offers a background threaded timer, that calls you back on a threadpool thread, and then you have async delegates, which are particularly useful with Web Services and so forth.

Using Tasks



Task-based Asynchronous Pattern (TAP)

- TAP is the recommended asynchronous design pattern for new development in .NET
- Based on the Task and Task<TResult> types in the System.Threading.Tasks namespace
 - used to represent asynchronous operations.



WHY TAP?

TAP is considered better than directly using threads because:

- **Higher-Level Abstraction**
 - The Task and async/await keywords are easier to understand and use
- **Improved Scalability:**
 - Leveraging asynchronous I/O operations.
- **Reduced Resource Consumption:**
 - Reduces the number of threads needed to handle asynchronous operations.
- **Simplified Error Handling:**
 - Allows exceptions to be propagated naturally through the call stack.
- **Better Composition:**
 - TAP allows asynchronous operations to be chained together using async/await.
- **Integration with Framework:**
 - Easy to use in conjunction with other asynchronous programming features such as asynchronous streams, parallel programming, and asynchronous event handling.

1. TAP provides a higher-level abstraction for asynchronous programming compared to directly working with threads. Instead of dealing with low-level threading constructs like Thread or ThreadPool, developers work with Task and async/await keywords, which are easier to understand and use.
2. Improved Scalability: TAP allows for better scalability by leveraging asynchronous I/O operations. Instead of blocking threads while waiting for I/O-bound operations (such as network requests or file I/O) to complete, TAP allows threads to be freed up to handle other tasks, improving the overall throughput of the application.
3. Reduced Resource Consumption: TAP minimizes resource consumption by reducing the number of threads needed to handle asynchronous operations. Asynchronous operations initiated with TAP can be multiplexed over a smaller number of threads, leading to more efficient use of system resources.
4. Simplified Error Handling: TAP simplifies error handling by allowing exceptions to be propagated naturally through the call stack. When using

async/await, exceptions thrown by asynchronous operations are automatically captured and rethrown when the Task is awaited, making it easier to handle errors in a structured and consistent manner.

5. Better Composition: TAP supports composability, allowing asynchronous operations to be chained together using async/await. This enables developers to write clean and concise code that expresses complex asynchronous workflows in a sequential and readable manner.
6. Integration with Framework: TAP is deeply integrated with the .NET framework and libraries, making it easy to use in conjunction with other asynchronous programming features such as asynchronous streams, parallel programming, and asynchronous event handling.

Overall, TAP provides a modern and efficient approach to asynchronous programming in .NET, offering benefits in terms of scalability, resource utilization, error handling, and code readability compared to directly using threads

Synchronous = **SLOW**

```
public static void PrepAndLaunchRocket() {  
    var watch = new System.Diagnostics.Stopwatch();  
    watch.Start();  
  
    MoveRocketToLaunchpad();  
    AddPayload();  
    LoadOxygen();  
    LoadHydrogen();  
    IgniteFuel();  
    Blastoff();  
  
    watch.Stop();  
    Console.WriteLine(  
        $"Execution Time: {watch.ElapsedMilliseconds}ms");  
}  
  
public static void MoveRocketToLaunchpad() {  
    Thread.Sleep(8000);  
    Console.WriteLine("Rocket on Launchpad");  
}  
...  
public static void Blastoff() {  
    Thread.Sleep(500);  
    Console.WriteLine("Rocket Launched!");  
}
```



Asynchronous
=
FAST
...
BUT
Uncoordinated

```
public static void PrepAndLaunchRocket() {
    var watch = new System.Diagnostics.Stopwatch();
    watch.Start();

    Task task1 = MoveRocketToLaunchpad();
    Task task2 = AddPayload();
    Task task3 = LoadOxygen();
    Task task4 = LoadHydrogen();
    Task task5 = IgniteFuel();
    Task task6 = Blastoff();

    Task.WaitAll(task1, task2, task3, task4, task5, task6);

    watch.Stop();
    Console.WriteLine(
        $"Execution Time: {watch.ElapsedMilliseconds}ms");
}

public static async Task MoveRocketToLaunchpad() {
    await Task.Run(() => {
        Thread.Sleep(8000);
        Console.WriteLine("Rocket on Launchpad");
    });
}

public static async Task Blastoff() {
    await Task.Run(() => {
        Thread.Sleep(500);
        Console.WriteLine("Rocket Launched!");
    });
}
```

**Controlled
Asynchronous
=**
**Compromised
Speed**
...
BUT
Coordinated

```
public static void PrepAndLaunchRocket() {
    var watch = new System.Diagnostics.Stopwatch();
    watch.Start();

    Task task1 = MoveRocketToLaunchpad();
    await task1;
    Task task2 = AddPayload();
    Task task3 = LoadOxygen();
    Task task4 = LoadHydrogen();
    Task task5 = IgniteFuel();
    Task task6 = Blastoff();
    await task6;

    watch.Stop();
    Console.WriteLine(
        $"Execution Time: {watch.ElapsedMilliseconds}ms");
}

public static async Task MoveRocketToLaunchpad() {
    await Task.Run(() => {
        Thread.Sleep(8000);
        Console.WriteLine("Rocket on Launchpad");
    });
}
```

Returning data

```
public static void PrepAndLaunchRocket() {
    ...
    Task<string> task6 = Blastoff();
    string message = await task6;
    Console.WriteLine(message);
    ...
}

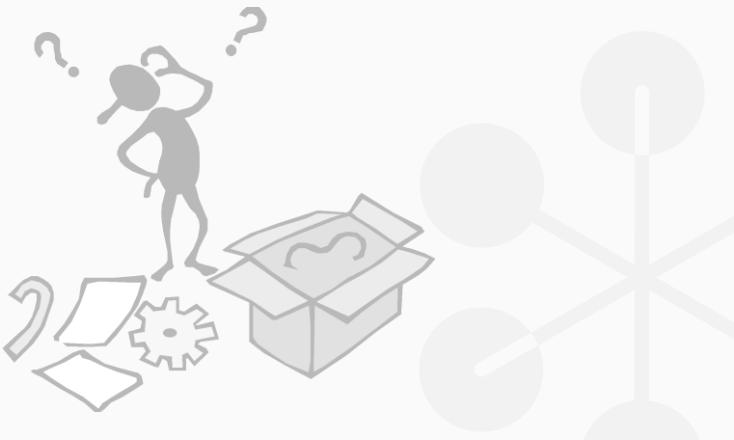
public static async Task<string> Blastoff()
{
    return await Task<string>.Run(() => {
        Thread.Sleep(8000);
        Console.WriteLine("Rocket on Launchpad");
        return "Houston, we have a lift off!";
    });
}
...
```

The generic type can be omitted here

But **not** here

Using Tasks

Demonstration



Look at the RocketLaunch demo to see a demonstration of using Tasks

Task Continuation (Chaining)

- Used to specify what should happen after a task completes
 - either successfully or with an exception.
- It provides a way to chain multiple tasks together
 - Where one task starts executing after another task finishes
 - Or to execute different logic based on the outcome of a task

Task.Continuewith()

```
Task<string> fetchDataTask =
    FetchDataAsync(
        "https://jsonplaceholder.typicode.com/todos");

    // Continue with different tasks based on
    // fetch task result
    await fetchDataTask.Continuewith(async task =>
{
    if (task.IsFaulted) ...
```

Check out the TaskContinuation Demo

The Stock Ticker App - How to use Asynchronous Methods

- Suppose we have a simple application that needs to fetch Stock Ticker information from a web API.
- Without asynchronous programming:
 - fetching data synchronously would block the main thread
 - the UI would freeze and making the application unresponsive.
- By using asynchronous programming with TAP, we can fetch data asynchronously without blocking the main thread.

The example demonstrates how to use asynchronous methods to improve the responsiveness and scalability of an application.

Suppose we have a simple application that needs to fetch stock exchange data from a web API. Without asynchronous programming, fetching data synchronously would block the main thread, causing the UI to freeze and making the application unresponsive. By using asynchronous programming with TAP, we can fetch data asynchronously without blocking the main thread.

Stock Ticker Company and RootObject classes

```
public class Company {  
    [JsonProperty(PropertyName = "1. symbol")]  
    public string Symbol { get; set; }  
  
    [JsonProperty(PropertyName = "2. name")]  
    public string Name { get; set; }  
  
    [JsonProperty(PropertyName = "3. type")]  
    public string Type { get; set; }  
  
    [JsonProperty(PropertyName = "4. region")]  
    public string Region { get; set; }  
  
    [JsonProperty(PropertyName = "5. marketOpen")]  
    public string MarketOpen { get; set; }  
  
    [JsonProperty(PropertyName = "6. marketClose")]  
    public string MarketClose { get; set; }  
  
    [JsonProperty(PropertyName = "7. timezone")]  
    public string Timezone { get; set; }  
  
    [JsonProperty(PropertyName = "8. currency")]  
    public string Currency { get; set; }  
  
    [JsonProperty(PropertyName = "9. matchScore")]  
    public string MatchScore { get; set; }  
}  
  
public class RootObject {  
    public List<Company> BestMatches { get; set; }  
}
```

The example fetches real-time stock ticker data from an online API asynchronously using the Task-based Asynchronous Pattern (TAP) in C#.

1. Define the Customer and RootObject classes: Define classes that map to the JSON structure returned by the service

2. Stock Ticker Service Interface: An interface IStockTickerService with an asynchronous method GetStockDataAsync to fetch stock ticker data.

3. Stock Ticker Service Implementation: Implementation of the IStockTickerService interface to fetch real-time stock ticker data from the online API asynchronously using HttpClient.

4. Use the Stock Ticker Service: Use the IStockTickerService to fetch stock ticker data asynchronously and process/display it.

Stock Ticker Service Interface

```
public interface IStockTickerService {  
    Task<List<Company>> GetStockDataAsync(string symbol);  
}
```



The example fetches real-time stock ticker data from an online API asynchronously using the Task-based Asynchronous Pattern (TAP) in C#.

1. Define the Customer and RootObject classes: Define classes that map to the JSON structure returned by the service

2. Stock Ticker Service Interface: An interface IStockTickerService with an asynchronous method GetStockDataAsync to fetch stock ticker data.

3. Stock Ticker Service Implementation: Implementation of the IStockTickerService interface to fetch real-time stock ticker data from the online API asynchronously using HttpClient.

4. Use the Stock Ticker Service: Use the IStockTickerService to fetch stock ticker data asynchronously and process/display it.

Stock Ticker Service implementation

```
using Newtonsoft.Json;

public class StockTickerService : IStockTickerService {
    string APIKey = "????????????????????";
    string QUERY_URL = "";

    private readonly HttpClient httpClient;

    public StockTickerService() {
        httpClient = new HttpClient();
    }

    public async Task<List<Company>> GetStockDataAsync(string symbol) {
        QUERY_URL = $"https://www.alphavantage.co/query?" +
            $"function=SYMBOL_SEARCH&keywords={symbol}&apikey={APIKey}";
        Uri queryUri = new Uri(QUERY_URL);
        HttpResponseMessage response = await httpClient.GetAsync(queryUri);

        if (response.IsSuccessStatusCode)
        {
            string jsonResponse = await response.Content.ReadAsStringAsync();
            var result =
                JsonConvert.DeserializeObject<RootObject>(jsonResponse);
            return result.BestMatches;
        }
        else {
            throw new Exception($"Failed to fetch stock data.");
        }
    }
}
```

The example fetches real-time stock ticker data from an online API asynchronously using the Task-based Asynchronous Pattern (TAP) in C#.

1. Define the Customer and RootObject classes: Define classes that map to the JSON structure returned by the service

2. Stock Ticker Service Interface: An interface IStockTickerService with an asynchronous method GetStockDataAsync to fetch stock ticker data.

3. Stock Ticker Service Implementation: Implementation of the IStockTickerService interface to fetch real-time stock ticker data from the online API asynchronously using HttpClient.

4. Use the Stock Ticker Service: Use the IStockTickerService to fetch stock ticker data asynchronously and process/display it.

Using the Stock Ticker Service

```
public class Program {
    private static readonly IstockTickerService stockTickerService =
        new StockTickerService();

    public static async Task Main(string[] args) {
        try {
            string symbol = "tesco"; // stock symbol
            Console.WriteLine($"Fetching stock data for symbol {symbol}...");

            List<Company> companies =
                await stockTickerService.GetStockDataAsync(symbol);

            Console.WriteLine($"Stock data for symbol {symbol}:");
            foreach (var cmp in companies) {
                Console.WriteLine($"Symbol: {cmp.Symbol}, " +
                    $"Name: {cmp.Name}, Type: {cmp.Type}, " +
                    $"Rgn: {cmp.Region}, Mkt Open: {cmp.Marketopen}, " +
                    $"Mkt Close: {cmp.MarketClose}, TZone: {cmp.Timezone}, " +
                    $"Curr: {cmp.Currency}, Match Score: {cmp.MatchScore}");
            }
        }
        catch (Exception ex) {
            Console.WriteLine($"Error: {ex.Message}");
        }
    }
}
```

The example fetches real-time stock ticker data from an online API asynchronously using the Task-based Asynchronous Pattern (TAP) in C#.

1. Define the Customer and RootObject classes: Define classes that map to the JSON structure returned by the service

2. Stock Ticker Service Interface: An interface IStockTickerService with an asynchronous method GetStockDataAsync to fetch stock ticker data.

3. Stock Ticker Service Implementation: Implementation of the IStockTickerService interface to fetch real-time stock ticker data from the online API asynchronously using HttpClient.

4. Use the Stock Ticker Service: Use the IStockTickerService to fetch stock ticker data asynchronously and process/display it.



Processing Tasks as they Complete

Task.WhenAny

```
public static async Task<List<string>> GetTypicodeData(string[] apiUrls)
{
    // List to store the results from each API call
    List<string> results = new List<string>();

    // Create tasks to fetch data asynchronously from each API
    List<Task<string>> fetchDataTasks = new List<Task<string>>();

    // Initiate multiple Tasks at the same time
    foreach (string apiUrl in apiUrls)
    {
        fetchDataTasks.Add(FetchDataAsync(apiUrl));
    }

    // Process tasks as they complete (rather than their starting order)
    while (fetchDataTasks.Count > 0)
    {
        // Await completion of any task
        Task<string> completedTask = await Task.WhenAny(fetchDataTasks);

        // Remove completed task from list
        fetchDataTasks.Remove(completedTask);
        // Await the result of the completed task
        string result = await completedTask;

        // Process the result
        results.Add(result);
    }
    return results;
}
```

```
// Process tasks as they complete
while (fetchDataTasks.Count > 0)
{
    //Awaits a call to WhenAny to identify the first task in the collection that has
    finished its download.
    Task<string> completedTask = await Task.WhenAny(fetchDataTasks);

    // Remove the completed task from the list
    fetchDataTasks.Remove(completedTask);

    // Await the result of the completed task, which is returned by a call to
    // FetchDataAsync. The completedTask variable is a Task<TResult> where TResult is a
    // string. The task is
    // already complete, but you await it to retrieve the data from the downloaded
    // website
    string result = await completedTask;

    // Process the result
    results.Add(result);
}
return results;
}

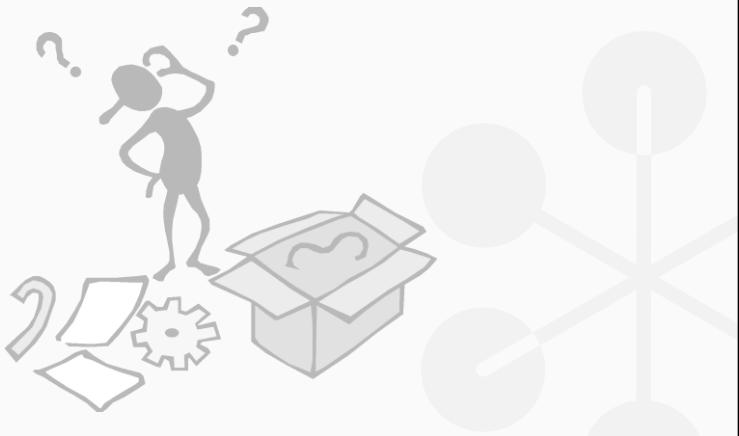
// Method to fetch data asynchronously from a web API
public static async Task<string> FetchDataAsync(string apiUrl)
{
    using (HttpClient httpClient = new HttpClient())
    {
```

```
 HttpResponseMessage response = await httpClient.GetAsync(apiUrl);

if (response.IsSuccessStatusCode)
{
    return result;
}
else
{
    throw new Exception($"Failed to fetch data from {apiUrl}. Status code:
{response.StatusCode}");
}
}
```

Using Task.WhenAny

Demonstration



Look at the ProcAsyncTasksAsTheyComplete demo to see a demonstration of using Task.WhenAny

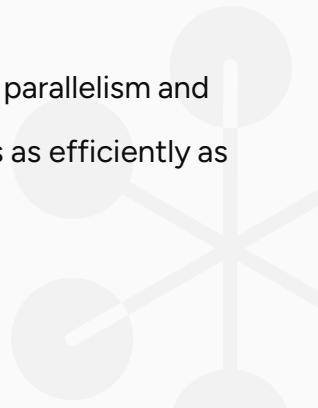
Cancelling Tasks

- To cancel an async task use:
- `CancellationTokenSource.Token`
- Pass instance of token as parameter to any await calls
- Invoke token's `Cancel` function
 - Any running tasks that have the token will be cancelled
 - On triggering Cancel code will need to wait for the cancellation to take place
 - For further details see: [Cancel a list of tasks - C# | Microsoft Learn](#)
- Use `CancelAfter` to cancel after a period of time.

Task Parallel Library (TPL)

TPL:

- A set of types and APIs in the System.Threading and System.Threading.Tasks namespaces
- Aim is to boost productivity by making it easier to add parallelism and concurrency to code.
- TPL automatically scales to use all available processors as efficiently as possible
- Handles:
 - ThreadPool thread scheduling
 - Cancellation support
 - State management
 - Partitioning of work



Parallel.For and Parallel.ForEach

- Used to execute a for loop in parallel
- It distributes the iterations of the loop across multiple threads
- Allows concurrent execution of loop iterations
- Takes advantage of multi-core processors to improve performance.

```
// Perform some operation on each element of the array using ParallelFor
Parallel.For(0, 11, i =>
{
    // Generate the first 10 powers of i
    for (int j = 1; j < 11; j++)
    {
        numbers[i, j] = (long)Math.Pow(i, j);
        Console.WriteLine($"{i,2} raised to the power of {j,2} is {numbers[i, j],12}");
    }
});
```

```
// Perform an operation on each array element using ParallelForEach
int[] seedNums = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
Parallel.ForEach(seedNums, i =>
{ ... }
```

Thread-Local and Partition-Local Variables

	Thread-Local	Partition-Local
Scope	Unique to each individual thread. Each thread has its own copy of the variable, and changes made to the variable in one thread do not affect its value in other threads.	Specific to each partition of work created by a parallel loop. Each partition of the loop has its own instance of the variable, changes made to the variable in one partition do not affect its value in other partitions.
Lifetime	Exist for the entire duration of the thread's execution.	exist only within the scope of the partition of work they are associated with. Once the partition completes its execution, the variable is typically no longer accessible.
Usage	Useful when you need to maintain separate instances of a variable for each thread, such as maintaining per-thread counters or accumulators in parallel loops.	Useful when you need to maintain separate instances of a variable for each chunk of work in a parallel loop, such as maintaining per-partition accumulators or temporary storage.
Declaration	Use the <code>ThreadLocal<T></code> class or the <code>[ThreadStatic]</code> attribute.	Variables typically created implicitly within the body of a parallel loop. The compiler ensures that each partition has its own instance of the variable

Thread-local and partition-local variables are mechanisms for managing data in parallelized code, particularly within parallel loops like `Parallel.For`. While they share similarities, they differ in terms of scope and lifetime.

While both thread-local and partition-local variables serve the purpose of isolating data in parallelized code, they differ in terms of their scope, lifetime, and usage. Thread-local variables are specific to individual threads, while partition-local variables are specific to partitions of work within parallel loops. Understanding the distinction between these two types of variables is important for writing efficient and correct parallel code in C#.

Hands on Lab

Asynchronous Programming and Concurrency

Objective:

Your goal is to experiment with different asynchronous programming scenarios. Multiple processes, Multiple CPU cores, using a queue

You will use a program that reads the contents of a file that contains a large number of words and generates the most popular stems of 2 to n characters long. You will create multiple versions of the code that split the searching into separate tasks where each task works on a separate stem size (2 chars, 3 chars, etc.). You will investigate what happens if:

- Each stem size is allocated its own process
- You imagine the computer has 2 CPUs and you split the work between them with each being given a range of stem sizes to handle.
- You stick with the 2 CPU idea but use a queue of all the stem sizes for the two processes to work through.

Summary

Now Know:

- Asynchronous programming in C#
- How to effectively use Tasks and Task Parallel Library (TPL) for asynchronous operations
- How to manage concurrency and understand best practices to avoid race conditions and deadlocks
- Asynchronous Programming Patterns
- Task-based Asynchronous Patterns (TAP)
- Threading
- Task Parallel Library (TPL)

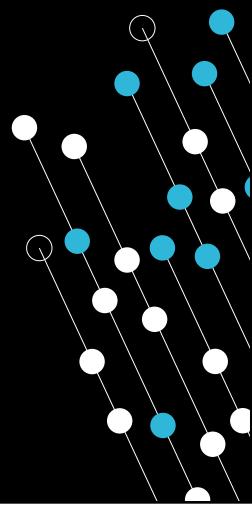


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Introduction to ASP.NET Core Web API Core MVC



Session Objectives

Gain an overview of an MVC application



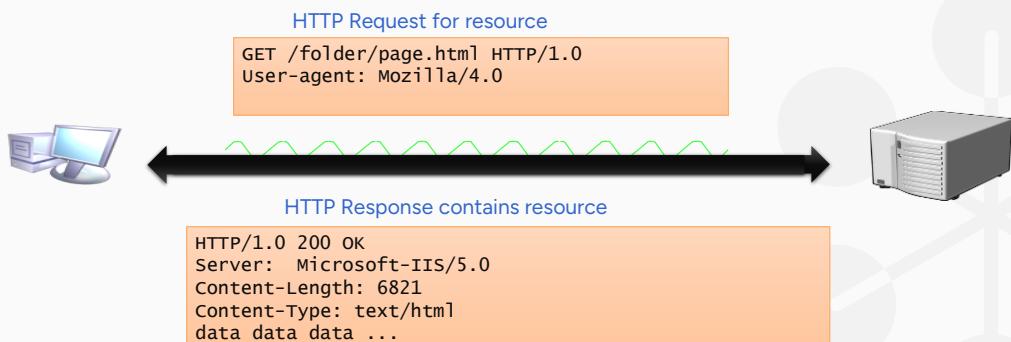
Session Content

- HTTP
- The MVC Pattern
- Models, Views, and Controllers
- ASP.NET MVC Conventions
- MVC Project structure
- Visual Studio support



Hypertext Transfer Protocol (HTTP)

- HTTP is an Application Level Protocol
 - A request and response protocol
 - Each request is independent from any other



HTTP is stateless. This means that each request for a resource from the Web server, such as that for an HTML page, is seen as being completely independent of any previous request.

The Web server doesn't store any information that associates the different requests, which in turn means that the Web server can scale well to deliver extremely large numbers of requests.

HTTP follows a simple request/response paradigm:

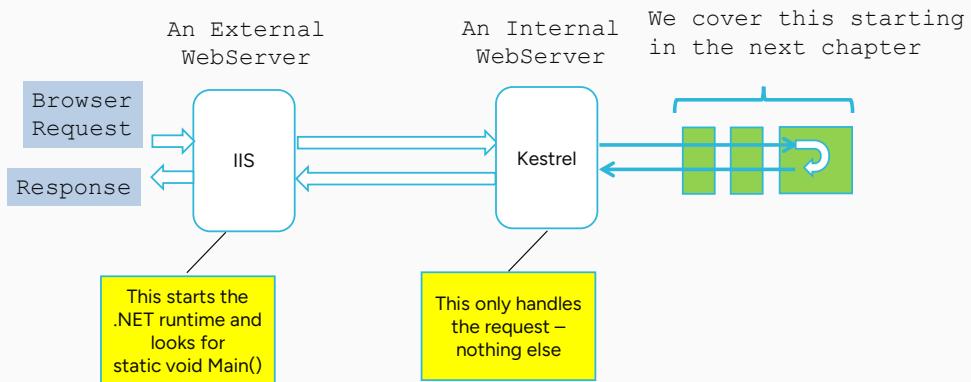
- 1.The connection is opened.
- 2.The browser sends an HTTP request for a resource.
- 3.The server sends an HTTP response containing the resource.
- 4.The connection is closed.

In HTTP version 1.1, the browser can request that the server keep the connection alive for a short while so that the browser can request another resource without opening another connection to the server.

Some of the advantages of ASP.NET (Core)

- It is no longer tied to IIS and Windows
- It has a modular request pipeline – so is typically very lightweight
- Is fully open-source and cross-platform
- It is extremely fast

The Pipeline



What is the MVC Design Pattern

- An architectural pattern that can be used when developing interactive applications
- Uses the principle of “Separation of Concerns”
 - Enhancing maintainability, extensibility, and testability
- Separates interaction logic into three areas:
 - Controller
 - Orchestrates the model and the view to produce the desired response
 - Model
 - Responsible for producing the data just for a view
 - View (not relevant in API setting)
 - Responsible for rendering the response
- Also:
 - Routing
 - Directs the user request to the correct controller and action

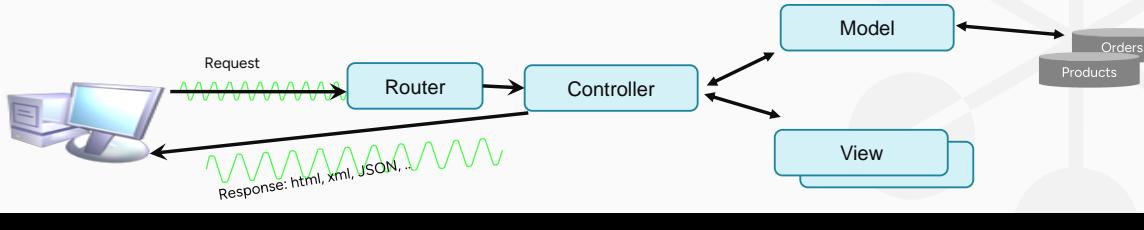


Model–view–controller (MVC) is a software architectural pattern for implementing user interfaces. It divides a given software application into three interconnected parts.

- A controller sends commands to the model to update and retrieve the model's state. It also selects the associated view to be returned.
- A model represents the information in the system. That information could be stored in a database, a file, or in memory. Its underlying store is abstracted way by the model, which provides a single interface upon which to interact with the data.
- A view renders the information from the model, it uses to generate the response to the user. In a RESTful API environment there is little or no need to include View functionality

MVC Architecture

- Controller
 - Translates user actions (http requests, Ajax calls, etc.) into application function calls, Works on the model, and selects the appropriate View based on preferences and Model state.
- Model
 - Probably better thought of as a ViewModel – ie. the extract of the business data specific for a view.
- View (**Not relevant in an API setting**)
 - Renders the content of the response (in Html).
 - CSS is used to turn this content into presentation



Routing

- Traditionally URLs map to file names
 - <http://www.mysite.com/default.html>
 - <http://www.mysite.com/Home/details.aspx>
- In MVC URLs are mapped to Actions (Methods) inside Controller (Classes)
 - <http://www.mysite.com/Home/Index> => HomeController.Index()
 - <http://www.mysite.com/Home/Index/5> => HomeController.Index(int id)
- In REST API applications this is generally done via "attribute routing":

```
[HttpGet("")]
[HttpGet("Home")]
[HttpGet("Home/Index")]
[HttpGet("Home/Index/{id?}")]
public IActionResult Index(int? id) {
    return GetMessage(id);
}
```

Before ASP.NET MVC, URLs in web application mapped to physical files at a disk location. So, for example if you had a URL

'<http://www.mysite.com/students/list.aspx>' it simply meant there was a list.aspx file in a 'students' folder. The URL had no other meaning. However, if this was an MVC site, the URL could look like '<http://www.mysite.com/student/>'. By convention the URL would map to the StudentController class with an action named Index.

The default route takes the first part of a Url path and assumes it contains the name of the controller, it assumes the second part of the URL, if it exists, is the name of an action in that controller. Allowing it to create the controller and invoke the action method inside that controller.

Model

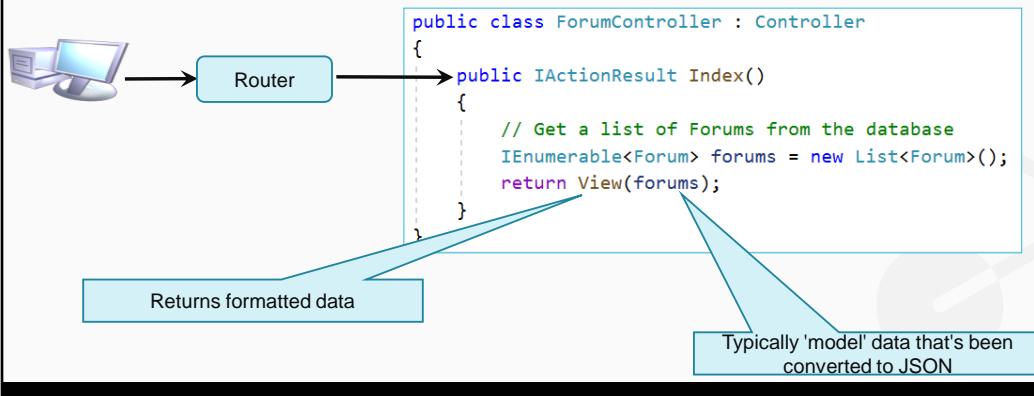
- We're going to need some data (in the Models folder):

```
public class Forum
{
    public int ForumId { get;set; }
    public string Title { get;set; }
}
```

- Note how we've added an Id – because we intend this to be stored in a database

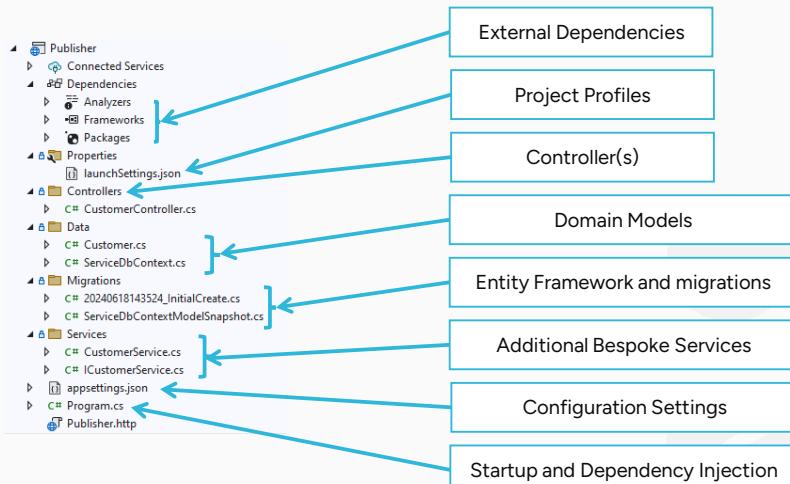
Controller

- Controller
 - Created and invoked by the Router when an HttpRequest is received
 - Methods inside a controller are referred to as **Actions**



The MVC controller is just a class that has inherited from the Controller base class. You add methods to this controller class to implement the functionality of the application. Methods are invoked by requests sent to the controller and are referred to as Actions. It is these Actions that control the interaction between the user and the application. A typical Action would create an appropriate model and convert the model to JSON before returning it.

Important folders/files in your application



ASP.NET MVC API projects are very “Conventional”.

Visual Studio assumes that all controllers are post fixed with Controller. E.g. `HomeController`. Therefore, any routes such as `/Home/Index` will be routed to the controller named `HomeController`.

It assumes that all views are located in a folder named after their controller. E.g. Views belonging to `HomeController` would be located in `Views\Home`. Views are assumed to be named after the action that returns them. E.g. An action named `Create` would return a view named `Create`.

Folder	Description
App_Data	Contains application data files including MDF files, XML files as well as other data store files. The App_Data folder is used by ASP.NET MVC to store an application's local database, which can be used for maintaining membership and role information,
Content	Contains .css files as well as image files and generic resources that define the appearance of views.
Controllers	This folder contains the controller class files.
Models	We rename this to <code>BuiltInModels</code> as it contains the Model and ViewModel classes required by the built-in authentication mechanism.
Scripts	This folder contains script files such as those required for jQuery or

Hands on Lab

A Quick Tour around ASP.NET MVC API Core

Objective:

Your goal is to make sense of the fundamentals of ASP.NET MVC API Core

You start out by creating a basic ASP.NET MVC API Core project and then explore how to go about adding a controller and giving it a set of Actions. You will explore how C# method overloading causes issues that can be overcome by adding routing via `HttpGet` attributes. You will test the applications by using the built in Swagger capabilities and also take a look at using Postman as an alternative.

Review

Gain an overview of an MVC application

- The MVC Pattern
- Routers, Controllers, Models, View-Models, Views and Areas
- MVC Project structure
- Visual Studio support



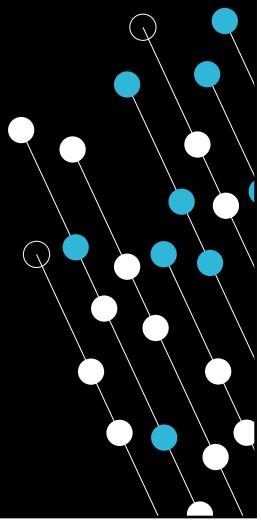


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

ASP.NET MVC API Core Dependency Injection



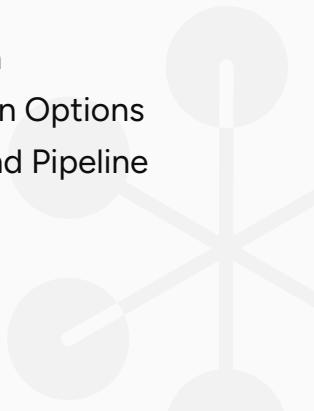
Session Objectives

Gain an understanding of how the Dependency Injection pattern is implemented and used in an MVC application



Session Content

- Dependency Injection
- Dependency Injection Framework in .NET Core
- Registering and Injection
- Injection of Configuration Options
- The Request Lifecycle and Pipeline



SOLID

You have already looked at the SOLID principles of good s/w:

- Single Responsibility Principle
- Open / Closed Principle
- Liskov's Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

Dependency Injection is so good that the whole of hosting, configuration, logging and ASP.NET Core is built using it.

197

You may also have come across **STUPID**

Singleton
Tight coupling
Untestability
Premature Optimization
Indescriptive Naming
Duplication

I think this is a case where the acronym preceded the meaning!

With Dependency Injection, always be very suspicious of the 'new' keyword – because with the 'new' keyword you are committing yourself to an implementation.

Benefits of Dependency Injection (DI)

- Stops High-Level modules being dependent on low-level modules
- Details should depend on abstractions
- Loose coupling
- None of the lifetime, creation or disposal issues clutter your business-focussed code
- Supports unit testing

Aspects of a DI Framework

There must be a globally available container

At startup specify an interface-to-implementation mapping

Scope

- Transient, Scoped and Singleton
- For other than Transient, consider thread-safety

Reflection of Constructor parameters

- Need a strategy for choosing a constructor when the constructor is overloaded

The ASP.NET Core DI Lifetimes

Transient

- Each time a dependent class requests a service, a new instance of the service class is instantiated
- Not required to be thread-safe, potentially less efficient

Scoped

- Produces a new service object for every new web-client request. Lasts for the whole request
- This means that if two classes, which are both used as part of the same request, need the same service, they will receive a single object shared between them. But only for the duration of that client request
- It is NOT a session, i.e. it does not keep data once the client request is completed

Singleton

- Produces a new object which lasts the lifetime of the whole application
- Must be thread-safe
- Suited to stateless services
- Potentially most performant

200

Beware of accidental sharing between objects of different lifetimes e.g. a transient object (which does not need to be thread-safe), if it is created by an object with singleton lifetime then suddenly it will be working in a thread-unsafe environment.

Note that ‘Scoped’ means ‘The same Http Context’ – it absolutely does not mean ‘Session’ in the sense that web programmers would understand it.

Think of ‘Singleton’ as being a static variable at the outmost level of the application

Registering

- Dependencies are registered in Program.cs
- By specifying the scope and which concrete class should be used to satisfy which interface:

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

builder.Services.AddTransient<ITransient, TransientDependency>();
builder.Services.AddScoped<IScoped, ScopedDependency>();
builder.Services.AddSingleton<ISingleton, SingletonDependency>();
```

The DI system allows multiple registrations – then you get ‘Last One Wins’.

But another reason for multiple registrations is if eg you have a set of rules that need to be evaluated. Now you would certainly enter multiple registrations. On the consuming side you would need to specify
IEnumerable<IRule> rules

Then in your code, iterate around them all

Constructor Injection

- The most common arrangement is to use constructor injection :

```
public class ForumViewModel
{
    ILogging log;
    public ForumViewModel(ILogging log)
    {
        this.log = log;
    }
}
```



Action Injection

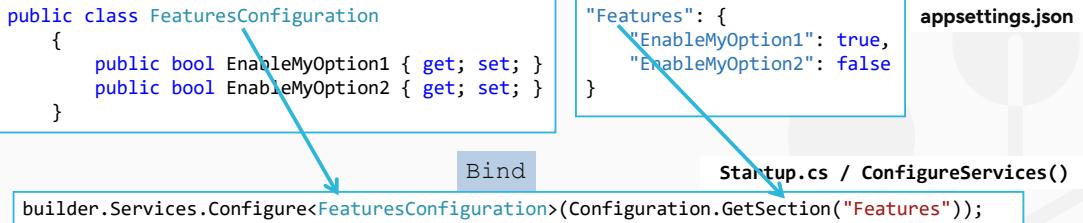
If you have a rarely-used and possibly expensive resource, instead of injecting every time can use Action Injection

```
public IActionResult Rare([FromServices]IActionInjection ai)  
{
```

- Only used if that specific action is invoked, not for other actions in the same controller

Configuration

- There is a built-in mechanism to bind text-specified config settings in appsettings.json to a strongly typed class:



Use

```
FeaturesConfiguration features;
public ForumViewModel(IOptions<FeaturesConfiguration> features) {
    this.features = features.Value;
    ...
}
```

You should not use regular DI to get configuration data.
Instead use the provided options pattern as shown

A type-unsafe way of reading config data

- There is an interface IConfiguration that allows you to read information from appsettings.json.
- This is type unsafe but is refreshed if you change the file and refresh the browser.

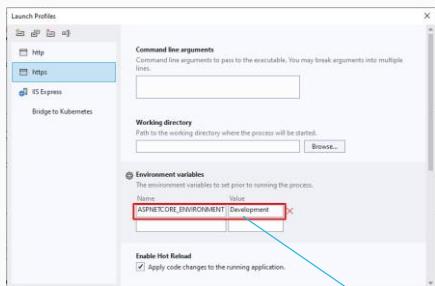
```
IConfiguration config;
public MyController(..., IConfiguration config)
{
    ...
    this.config = config;
}
...
config["Features:EnableMyOption1"]
```

• or

```
if (config.GetSection("Features").GetValue<bool>("EnableMyOption1")) {
    ...
} else {
    ...
}
```

Environment configuration

Set the environment variable



Create Environment-specific files

appsettings.json
appsettings.Development.json
appsettings.Staging.json
Program.cs

Appsettings.Development.json:
"Message": "Hello from appsettings.Development.json"

Appsettings.Staging.json:
"Message": "Hello from appsettings.Staging.json"

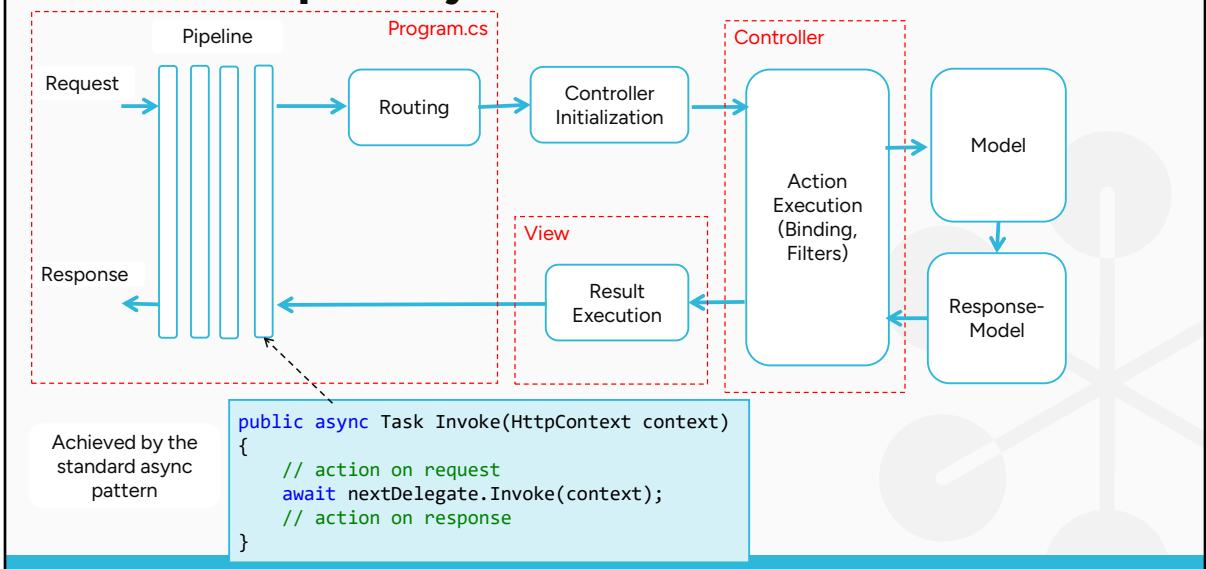
Add to Program.cs

```
Microsoft.Extensions.Configuration.ConfigurationManager config = builder.Configuration;
if (!app.Environment.IsDevelopment())
{
    app.UseSwagger();
    app.UseSwaggerUI();
}
Console.WriteLine(config["Message"]);
```

A blue arrow points from the 'config["Message"]' line in the code to the 'Message' key in the 'Appsettings.Development.json' section above.

Writes "Hello from appsettings.Development.json"

The MVC Request Cycle



The order in which the various pipeline elements are defined in program.cs file where the app object is being configured is the order through which each request will pass. i.e. the sequence of statements in Startup/Configure() is important. Contrast this with when services are added to the builder object's Services collection where the sequence is typically not important.

The Pipeline

- There are 3 types of pipeline component:
 - Run
 - These generate a response and short-circuit the life-cycle by returning and not calling the next one in the chain. Examples are the static file middleware or the Mvc middleware.
 - Use
 - These do some processing then pass on to the next in the chain. An example is authentication
 - Map
 - These conditionally send the request on to other middleware. An example would be routing for WebAPI requests

Hands on Lab

Dependency Injection, Scope and Configuration

Objective:

Your goal is to investigate how Dependency Injection and its configuration is done in ASP.NET applications

In this exercise you get to do Dependency Injection (DI) and learn how it is configured. You will also see the effect of the different DI scopes.

Review

Gain an understanding of how the Dependency Injection pattern is implemented and used in an MVC application

- Dependency Injection
- Dependency Injection Framework in .NET Core
- Registering and Injection
- Injection of Configuration Options
- The Request Lifecycle and Pipeline

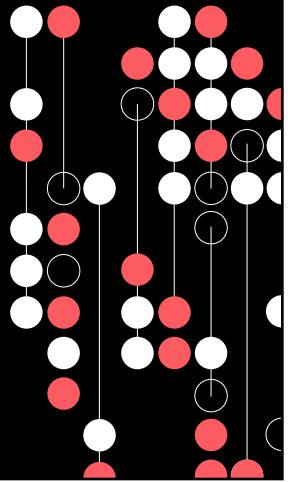


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

The Entity Framework



Session Objectives

To give an introduction to Entity Framework

Session Content

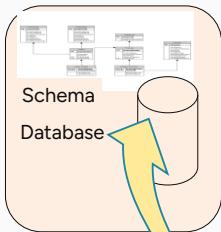
- The main features of Entity Framework
- Database / Model / Code First / Code First from Database
- Lazy / Eager / Explicit loading
- Linq To Entities
- Initialization and Migration

Why have an ORM at all?

Most applications use OO code and store in relational data hence there is a mis-match. To do-it-yourself you need to:

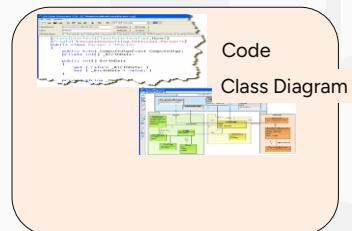
- Write the object model
- Write the plumbing code - synchronising, lazy loading, persistence, track changes, concurrency, validation, data type conversions, relationships and associations, vendor independence, make it a LINQ-provider.
- Typically, the DIY way consumes ~35% of project code & budget. With an ORM this typically drops to <10%.

Code First



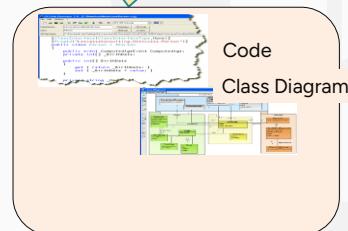
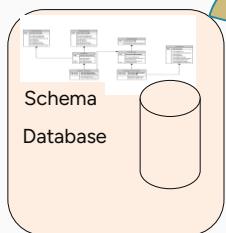
We use this process on
this course

Possible annotations
to control DDL



Code First From Database

Possible 'Buddy'
classes to add
annotations



To get started

EF favours convention, but almost everything can be modified by configuration

'NuGet'

Microsoft.EntityFrameworkCore.SqlServer

The minimum you need are:

- Properties NOT Fields!!
- An Id Property
- A Context
- What provider
- What database name, permissions etc
- What database tables will be exposed to the code

Code samples for getting started

Properties and Ids

```
public class Zoo {  
    public int ZooId { get; set; }  
    public string Name { get; set; }  
}
```

Context

```
public class ZooContext : DbContext {  
    public DbSet<Zoo> Zoos { get; set; }  
    public DbSet<Animal> Animals { get; set; }  
}
```

Using LINQToEntities

A regular 'Select'

```
using (var ctx = new NorthwindEntities()) {
    var customers = ctx.Customers.Where(c => c.Country == "Germany");
    foreach(var c in customers)
        Console.WriteLine(c.ContactName + ", " + c.City);
}
```

Change property and save

```
using (var context = new NorthwindEntities()) {
    var customer = context.Customers.First(c => c.CustomerID == "ALFKI");
    customer.City = "Berlin";
    context.SaveChanges();
}
```

LINQ

- Lambda-style

```
using (var ctx = new NorthwindEntities()) {  
    var customers = ctx.Customers.Where(c => c.Country == "Germany");  
    foreach(var c in customers)  
        Console.WriteLine(c.ContactName + ", " + c.City);  
}
```

- Query Expression Style

```
var customers = from c in ctx.Customers  
                 where c.Country == "Germany"  
                 select c;  
foreach(var c in customers)  
    Console.WriteLine(c.ContactName + ", " + c.City);
```

Lambda expressions

```
ctx.Customers.Where(c=>c.Country == "Germany")
```

The above is really a sort of shorthand for this

```
...
    var x = ctx.Customers.Where(GetABoolFromObject);
}
static bool GetABoolFromObject(Customer c) {
    return c.Country == "Germany";
}
```



Configuration

The screenshot shows a file explorer window with the following structure:

- QuickTour
- Connected Services
- Dependencies
- Properties
- wwwroot
- Areas
- Configuration
- Controllers
- Data
- Middleware
- Models
- Views
- appsettings.json
- Program.cs

The `appsettings.json` file is selected. The `Program.cs` file is also visible.

The `appsettings.json` file contains the following JSON configuration:

```
{  
  "ConnectionStrings": {  
    "DefaultConnection": "Server=\\SqlExpress;Database=MyDatabase;Trusted_Connection=True;MultipleActiveResultSets=true"  
  }  
}
```

The `Program.cs` file contains the following C# code:

```
var connectionString = builder.Configuration.GetConnectionString("DefaultConnection")  
  ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");  
builder.Services.AddDbContext<ApplicationContext>(options =>  
  options.UseSqlServer(connectionString));
```

A red arrow points from the highlighted line in the JSON to the corresponding line in the C# code. A callout bubble with the text "Note the double-slash in json files" points to the double-slash notation in the JSON connection string.

The name of the database connection string can be passed into the `DbContext` constructor.

Note that newlines are not permitted in the connection string – this is done only so they can fit onto the page

Context

```
public class ApplicationDbContext : IdentityDbContext
{
    public ApplicationDbContext(DbContextOptions<ApplicationDbContext> options)
        : base(options)
    {
    }

    protected override void OnModelCreating(ModelBuilder builder)
    {
        base.OnModelCreating(builder);
    }

    protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
    {
        base.OnConfiguring(optionsBuilder);
    }
}
```

Mandatory

Optional



Strategies for loading data

Lazy Loading

- Principally for collections, but can be used for large fields
- Loads the data on demand
- Suitable when many usages of the parent data will not require the detail of the child data

Eager Loading

- Suitable when the data is not huge and will very likely be needed shortly anyway

Explicit Loading

- Suitable when you want to be in control over what is loaded when

Loading data in code

- Lazy Loading

```
public class Zoo {  
    public virtual ICollection<Animal> Animals { get; set; }  
}
```

- Eager Loading

```
foreach (Zoo zoo in ctx.Zoos.Include(z=>z.Animals) ) {  
    ...  
}
```

- Explicit Loading

```
ctx.Entry(zoo).Collection(z => z.Animals).Load();
```



Managing changes to schema

- **Migration**

- Preserves data
- Needs to be actioned via the Package Manager Console

```
Enable-Migrations  
Add-Migration "Initial"  
  
Update-Database  
Or  
Update-Database -Script
```



```
↳ Migrations  
↳ 20190626140737_Initial.cs  
↳ ZooContextModelSnapshot.cs
```

Migration commands

Enable-Migrations:

- Enables Code First Migrations in a project
- Adds the Migrations folder to the Project
- Adds a configuration class to Migrations folder
- Add-Migration:
 - Creates a new migration
 - Scaffolds a migration script to apply pending model changes
 - Contains version number

Update-Database:

- Adds __MigrationHistory table to target database to track update history
- Entity Framework compares model version with version in database. Applies the changes if needed
- Get-Migrations:
- Displays the migrations that have been applied to the target database

When you deploy a completed web app and want to deploy the database alongside it, you will typically use Code First Migrations. When you create the publish profile that you use to configure settings for deploying from Visual Studio, you'll select a check box labelled Execute Code First Migrations (runs on application start). This setting causes the deployment process to automatically configure the application's appsettings.json file on the destination server so that Code First uses the MigrateDatabaseToLatestVersion initializer class.

Visual Studio doesn't do anything with the database during the deployment process while it is copying your project to the destination server. When you run the deployed application and it accesses the database for the first time after deployment, Code First checks if the database matches the data model. If there's a mismatch, Code First automatically creates the database (if it doesn't exist yet) or updates the database schema to the latest version (if a database exists but doesn't match the model).

Configuration

Using Attributes

```
[Required(ErrorMessage="Choose a forum for this thread")]
public int ForumID { get; set; }
[Column("OwnerID")]
public string UserID { get; set; }
```

Using the Fluent API

```
public class ApplicationDbContext : IdentityDbContext<ApplicationUser>
{
    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Forum>()
            .HasOptional(a => a.Threads)
            .WithOptionalDependent()
            .WillCascadeOnDelete(true);

        base.OnModelCreating(modelBuilder);
    }
}
```

When working with Code First, you define your model by defining your domain CLR classes. By default, the Entity Framework uses the Code First conventions to map your classes to the database schema. If you use the Code First naming conventions, in most cases you can rely on Code First to set up relationships between your tables based on the foreign keys and navigation properties that you define on the classes. If you do not follow the conventions when defining your classes, or if you want to change the way the conventions work, you can use the fluent API or data annotations to configure your classes so Code First can map the relationships between your tables.

IEnumerable / IQueryable

IEnumerable allows you to iterate collections

IQueryable does the iteration but also allows you to pass *commands* (rather) than *data* across your application

Beware of multiple unintended database accesses

The general recommendation is to pass `IList<>` (i.e. an evaluated `IEnumerable`) as your interface

```
public interface IEnumerable
{
    object Current { get; }
    bool MoveNext();
    void Reset();
}

public interface IQueryable : IEnumerable
{
    string Provider { get; }
}
```

Further Entity Framework

- Not covered in the course, but also present in the product:
 - Managing Inheritance
 - Using Stored Procedures
 - Script update of database
 - Managing Concurrency

231

EF has 2 out-of-the-box strategies for managing inheritance – Table-Per-Hierarchy (the default) and Table-Per-Type.



Hands on Lab

Introduction to the Entity Framework

Objective:

Your goal is to make sense of the fundamentals of the Microsoft Entity Framework. There are two labs (one of which is optional) which will walk you through how to take "Code First" and "Code First From Database" approaches to hooking C# code up to a SQL Server database.

You will learn how to take "Code First" and "Code First From Database" approaches to linking code to a SQL Server database. Steps will include:

- Installing appropriate packages using NuGet
- Define connection strings
- Create DbContext objects
- Define Model classes
- Perform Migrations to create and update databases
- Carry out CRUD operations against a database

Summary

Entity Framework is a World-Class ORM

- Works by convention – which can be changed by configuration
- Has strategies for loading data
- Schema change handled by Migration
- Handles inheritance, concurrency, stored procs etc.

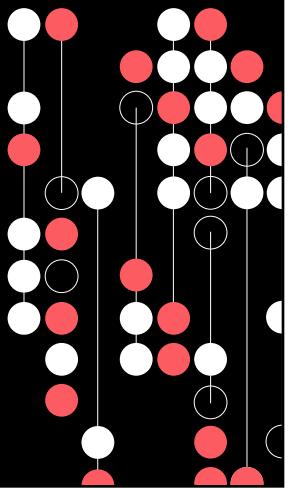


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Controllers and Actions



Session Objectives

Understand more about:

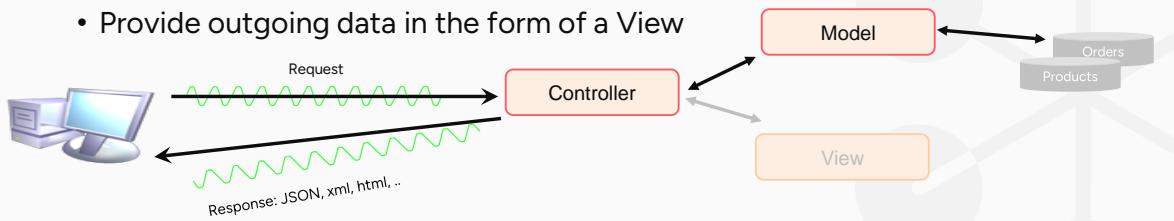
- The role of the Controller
- How to write and configure Actions to process requests

Session Content

- What is a Controller?
- Actions and Parameters
- Returning Results from Actions
- Asynchronous Actions
- Testing Web API Controllers

What is a controller?

- A controller is a class with the following responsibilities:
 - Handle an incoming request
 - Respond to user input
 - Make changes to the model in response to user input
 - Work with inbound data
 - Provide outgoing data in the form of a View



Implementing a Controller

- To mark a class as a Controller, it is sufficient to simply suffix its name with “Controller”
 - E.g. HomeController, ForumController
- It’s normal to inherit from the Controller class – gives access to a range of helper methods/properties

- Some we’ve used:
 - Ok() and NotFound()

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return Ok("Everything worked out fine!");
    }
}
```

All controllers must ultimately implement **IController** however, they usually inherit from either **ControllerBase** or **Controller**.

Both are abstract classes. **ControllerBase** implements **IController** directly and provides a rich API that includes properties such as **ValidateRequest**.

The **Controller** class actually inherits from **ControllerBase** and provides some additional behaviours such as ensuring that all public methods of the class are treated as actions.

Most controller classes inherit from the **Controller** class.

Web API Controllers

- Controllers:
 - Must inherit from ControllerBase
 - The ControllerBase class, which inherits from ControllerBase, contains features specific to MVC controllers
 - For controllers that only contain Web API actions, it is recommended to inherit directly from ControllerBase
 - Normally use attribute routing
 - Controller attribute indicates the URL prefix for the resource
 - Action attributes indicate the verb the action responds to, and anything that follows the prefix in the URL
 - By convention the route begins with "api/"
 - Note: [controller] within the route. [] is an alternative to { }

```
[Route("api/[controller]")]
[ApiController]
public class BuyerController : ControllerBase
{
    ...

    [HttpGet]
    public IEnumerable<Buyer> GetBuyers()
    {
        IEnumerable<Buyer>? buyers = _ctx.Buyers
        return buyers;
    }

    [HttpGet ("buyers/{id}")]
    public Buyer GetBuyerById(int id)
    {
        Buyer buyer = _ctx.Buyers
            .SingleOrDefault(b => b.Id == id);
        return buyer;
    }
}
```

Unlike previous versions of Web API, the name of the action method is no longer used to create routing conventions. Instead, verb attributes must be added to the methods.

The verb attributes specify anything to be added onto the controller's route prefix, for example an ID which is bound to the action's parameter.

The [ApiController] attribute

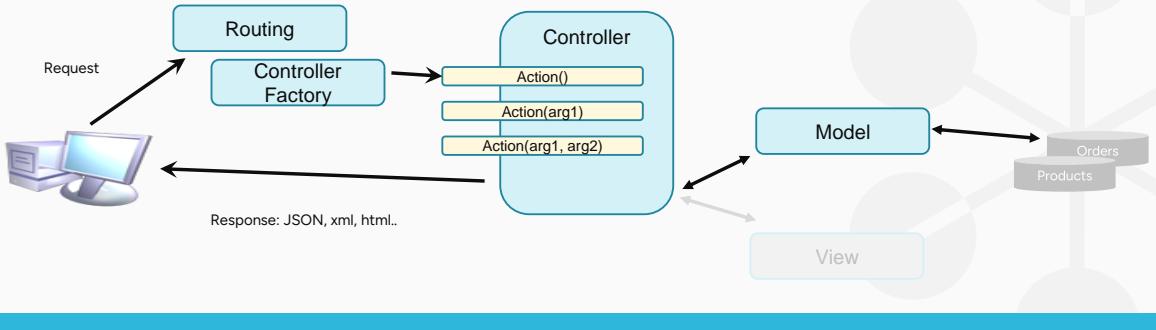
- The [ApiController] is *not* required
 - But it is recommended, because it changes the behaviour of the controller in ways that are normally desired in Web API
 - It prevents conventional routing from being used to reach the controller
 - It is normal that attribute routing is used for Web API controllers
 - It automatically checks the model state, and returns an error if the model is invalid
 - Equivalent of adding this into every action:

```
[HttpPost]
0 references
public ActionResult Post()
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }
}
```

241

Controllers Contain Actions

- Each inbound request is routed to a Controllers Action method
- The Controllers Action methods contains the code which process the request



The controller is perhaps the most important class in your application.

As requests are received and routes determined, the `MvcRouteHandler` class chooses the most appropriate controller for that route.

Part of the route will specify the name of the controller that is to be used and another part of the route will specify the action.

Client requests are not for web pages. Rather, they are requests for a resource or an action to be carried out. When a controller's action executes, it typically instantiates a model class and then passes into a view instance. The view is a file written in a format such as `ASPX` or `Razor`. A view engine then constructs the required `HTML` based upon the code in the view and the model data passed to it.

The resulting `HTML` (it needn't be `HTML`) is then returned from the action and passed onto the client.

Action Methods

- Action method can receive parameters
- Action methods can be overloaded

```
public class BuyerController : Controller
{
    ...
    [HttpGet("buyers")]
    public IEnumerable<Buyer> GetBuyers()
    {
        IEnumerable<Buyer>? buyers = _ctx.Buyers
        return buyers;
    }

    [HttpGet("buyers/{name}")]
    public Buyer GetBuyerById(string name)
    {
        IEnumerable<Buyer> buyers = _ctx.Buyers
            .Where(b => b.Name == name);
        return buyers;
    }
}
```

Actions are the endpoint of many requests and are analogous to both web page requests and web service method calls.

In both of those situations it is common to pass parameters. For example, a web page might be requested with an accompanying query string:

www.estateagents.com/buyers.aspx?id=7

However, the new notation is cleaner and easier to make sense of:

www.estateagents.com/api/Buyer/Patel

Action Methods and Route Values

- Arguments for action methods' parameters are called Route Values
 - Can be passed as part of the URL

```
public class PersonController  
{  
    public ActionResult Create(string name) {...}  
    // http://.../Person/Create?name=Mary  
  
    public ActionResult Create(string name, int age) {...}  
    // http://.../Person/Create?name=Mary&age=51
```

- Note how the arguments are passed using the standard HTTP query string format
- Any parameters not listed on the query string will be set to their default (Null, 0, false, or similar)

Default parameters

- Directly supported in the language

```
public class ProductController
{
    [HttpGet("List/{modelId}/{categoryId=4}",Name = "GetList")]
    public ActionResult List(int modelId, int categoryId) {...}
        // http://.../Product>List/123
        // Returns products in category 4

    // http://.../Product>List/123/5
    // Returns products in category 5
```

Where parameters are concerned, you must be able to handle a situation where one or more parameters have been omitted. Note when using routing the default value must go in the `HttpGet` (or `Route`) attribute and NOT as part of the method signature.

Attribute vs Conventional routing

- By default, ASP.NET Core API uses attribute routing, but you could switch to conventional routing (or both!) by altering Program.cs.

```
//Attribute routing
app.MapControllers();
//https://{{domain}}/{{controller}}/{{action}}/{{id?}}


//conventional routing
app.UseEndpoints(endpoints => {
    endpoints.MapControllers();
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{{controller=Home}}/{{action=Index}}/{{id?}}"); });
//https://{{domain}}/{{controller=Home}}/{{action=Index}}/{{id?}}
```

The default URL format in ASP.NET Core Web API using attribute routing typically looks like this:

`https://{{domain}}/{{controller}}/{{action}}/{{id?}}`

- **{domain}**: The domain where the application is hosted.
- **{controller}**: The name of the controller (minus the "Controller" suffix).
- **{action}**: The name of the action method.
- **{id?}**: An optional parameter, typically used for the resource identifier.

Example of Attribute Routing

```
[ApiController]
[Route("api/[controller]")]
public class ProductsController : ControllerBase {
    [HttpGet] public IActionResult GetAllProducts() {
        // Get all products logic
        return Ok(new List<string> { "Product1", "Product2" });
    }
    [HttpGet("{id}")]
    public IActionResult GetProductById(int id) {
        // Get product by id logic
        return Ok($"Product {id}");
    }
}
```

With the above setup, the following URLs would be available:

- `https://{{domain}}/api/products` (maps to `GetAllProducts`)
- `https://{{domain}}/api/products/{{id}}` (maps to `GetProductById` with an `id` parameter)

Action Methods with Complex Parameter Types

- If an action takes a parameter whose type is a class
 - Each property in turn is matched to the route values
- For this to work:
 - The class must have a parameterless constructor
 - The properties must have public setters

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}
```

```
[HttpPost]
public ActionResult Create(Person person) {...}
// http://.../Person/Create
// The "person" parameter must be passed in the request body
```

- This is called **Model Binding**

Action Return Types (for API apps)

- With the default media type formatters configured, based on the return type of your action:
 - Fundamental types, and some other types such as DateTime, are returned to browser as plain text
 - Most other types, e.g. application classes, are serialised as JSON
- You can get more control over the response by returning an ActionResult< TValue >
 - Enables you to specify the status code to be sent to the browser

```
[HttpGet("{id}")]
0 references
public ActionResult<Person> Get(int id)
{
    if (id >= Person.People.Length)
    {
        return NotFound();
    }
    return Person.People[id];
}
```

More About ActionResult

- When your method's return type is ActionResult, you can return:
 - Any class that inherits from ActionResult. Common examples are:
 - NotFoundResult (status code 404)
 - BadRequestResult (status code 400)
 - OkResult (status code 200)
 - OkObjectResult (status code 200, and also returns a serialised object)
 - You can create these objects yourself, but it's usually more convenient to use helper methods to create them:
 - NotFound() returns a NotFoundResult
 - BadRequest() returns a BadRequestResult
- return result; is a shortcut for return ok(result);
 - This shortcut is not available when the data type of the variable being returned is an interface

```
[HttpGet("{id}")]
0 references
public ActionResult<Person> Get(int id)
{
    if (id >= Person.People.Length)
    {
        return NotFound();
    }
    return Person.People[id];
}
```

Implicit Action Results

- You are not forced to return data as an ActionResult
- Methods returning non-ActionResult types are implicitly returned as ContentResult
 - ContentResult requires a string. MVC will call ToString on the returned value.

```
public string Detail(int ID)
{
    AdventureWorksRepository AWRep = new AdventureWorksRepository();

    var product = AWRep.GetProduct(ID);

    return product.Name;
}
```

It is not necessary to return ActionResults from actions.

You may for example wish to return a **double** or **int** type. In this case, ASP.NET MVC will implicitly return a **ContentResult** initialised with a **string** representation of your return value.

In fact, ASP.NET MVC calls **ToString** using the **InvariantCulture** on your return value and passes this into the **ContentResult**.

Data Transfer Objects

- It is possible to use Web API to serialise data directly from Entity Framework
- However, it's more common to create a Data Transfer Object specifically to be used with Web API
 - So that the return type of your actions might be `ActionResult<MyModelDTO>`
- This has a number of benefits:
 - We can choose exactly which properties to include
 - Can prevent sharing too much data
 - We can change the shape of the data
 - We can add extra attributes specifically for this API
 - We can remove circular references, which can prevent objects from being serialised correctly
 - (e.g. a Forum class that contains a collection of Threads, and each Thread links back to a Forum)

Model Binding

- There are three different places in the Request where a client can put data:

```
POST http://localhost:8602/api/person/12345?city=London HTTP/1.1
```

```
Content-Type: application/json
```

```
Content-Length: 24
```

```
{"Name": "John", "Age": 36}
```

```
public class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

[HttpPost("{id}")]
public void Post(int id, string city, Person person)
{}
```

- So long as the [ApiController] attribute has been used:

- Data before the ? in the URL is part of the route
 - Included in the route template parameter to [HttpPost]
- Primitive data types come from the query string by default
 - The query string is the part that comes after the ? in the URL
- Complex data types, e.g. classes, come from the body of the request

The [ApiController] attribute is needed to get the behaviour described on this slide. This is desirable, because it ensures that the place the model binder looks for its data corresponds with the most common parts of the request for clients to put that data.

If the [ApiController] attribute has not been used, the default place where the model binder searches for data is different, and it may be necessary to manually set the data source, using the attributes shown on the next slide.

Overriding the Model Source

- Attributes can be applied to individual parameters to change their source from the defaults:
 - [FromBody]
 - [FromQuery]
 - [FromRoute]

```
POST http://localhost:8602/api/person/12345?city=London&name=John&age=36 HTTP/1.1
```

```
[HttpPost("{id}")]
0 references
public void Post(int id, string city, [FromQuery] Person person)
{
```

- Note that only a single parameter is allowed to come from the request body
 - Either one single parameter which comes from the body by default
 - Or one single parameter with the [FromBody] attribute

More on Data Transfer Objects

- We have already seen how DTOs can be used as the return type from Actions
 - This means that the body of the Response will be a DTO
- But you can also use a Data Transfer Object as the parameter types
 - This means that the body of the Request will be a DTO
- Allows you to specify exactly which properties should be included in the Request
 - As well as ensuring that data you aren't expecting won't get bound to the model
 - Can prevent an "over posting attack"

Async and Await Action methods

- Asynchronous Actions are permitted
 - Cons (when not to use).
 - Simple or short running operations
 - Simplicity and testability are important
 - Operations are CPU-bound rather than IO-bound
 - Pros (when using should be considered)
 - Testing shows that blocking operations are bottlenecking site performance
 - Parallelism is more important than simplicity of code
 - Operations are IO-bound rather than CPU-bound
- Use for Action methods that call long running external processes:
 - Credit card validation
 - Web Service calls
 - Database calls
 - Anything that can block

When an action has to perform a long running process, not only does the client experience a delay in response it is also holding onto a thread from the thread pool.

By writing asynchronous actions, threads can be returned to the thread pool during long running processes. E.g. Action is awaiting response from a web service.

When the action is ready to continue, it can reacquire a thread from the thread pool and continue.

The client process will see no improvement in response time. However, the application will permit greater throughput of requests because there is less blocking in the thread pool.

Asynchronous Controller Actions

To enable an `async/await` Action method:

- Mark Action method with the `async` keyword
- Wrap the return type in `Task<returntype>`
- Call long running methods asynchronously in method body
- Use `await` keyword to pause execution and wait for results
 - Freed up the thread to be used elsewhere

```
public class AsyncDemoController : Controller
{
    private TrainingContext db = new TrainingContext();

    // GET: AsyncDemo/Details/5
    public async Task<ActionResult> Details(int? id)
    {
        if (id == null)
        {
            return new HttpStatusCodeResult(HttpStatusCode.BadRequest);
        }
        Student student = await db.Students.FindAsync(id);
        if (student == null)
        {
            return HttpNotFound();
        }
        return View(student);
    }
}
```

You can avoid performance bottlenecks and enhance the overall responsiveness of your application by using asynchronous programming. However, as has been highlighted in a previous session, traditional techniques for writing asynchronous applications can be complicated, making them difficult to write, debug, and maintain.

.NET supports a simplified approach to overcoming the problem, namely `async` programming, that leverages asynchronous support. The compiler does the difficult work that the developer used to do, and your application retains a logical structure that resembles synchronous code. As a result, you get all the advantages of asynchronous programming with a fraction of the effort.

Testing WebAPI Controllers

- When performing integration testing on WebAPI controllers, using a browser is of limited use
 - You can type a URL into the address bar
 - A GET request is sent, no option to use any other verbs
 - No control over request body
- The body of the response is shown in the browser
 - You can access response headers in developer tools in most web browsers
- A dedicated tool for testing RESTful APIs is much more useful
 - One common tool is Postman

Using Postman to Test WebAPI Controllers

- Example Postman screen:
- At the top is the verb (POST) and URL
- Below that is the Request
 - Currently showing the request body, which you can change
 - Can change to see other data e.g. headers, which you can also change
- At the bottom is the Response
 - Currently showing the response body
 - Can change to see other response data e.g. headers

The screenshot shows the Postman application interface. At the top, it displays a 'POST' method and the URL 'https://localhost:44314/api/Person?city=London'. The 'Body' tab is selected, showing a JSON payload:

```
1 * {
2     "Name": "John",
3     "Age": 36
4 }
```

Below the request, the response is shown in the 'Body' tab under 'JSON' format:

```
1 {
2     "personId": 123,
3     "name": "John",
4     "age": 36,
5     "city": "London"
6 }
```

At the bottom of the interface, status information is displayed: Status: 201 Created, Time: 151 ms, Size: 286 B.

Hands on Lab

Controllers and Actions

Objective:

Your goal is to add an ASP.NET API project that uses the Model View Controller (MVC) pattern to a Microsoft Visual Studio Estate Agent solution that provides CRUD capabilities to a SQL Server database's Buyers table

- You will create a microservice for "buyer" information. The microservice should be created from an ASP.NET **MVC** API template and allow a consumer of the service to:
 - Retrieve a list of all buyers.
 - Retrieve a buyer by their id.
 - Retrieve a buyer by their name.
 - Add new buyers.
 - Delete existing buyers.
 - Update existing buyers.

Summary

Know more about the configuration and use of Controllers and Actions

- What is a Controller?
- Actions and Parameters
- Returning Results from Actions
- Asynchronous Actions
- Testing Web API Controllers



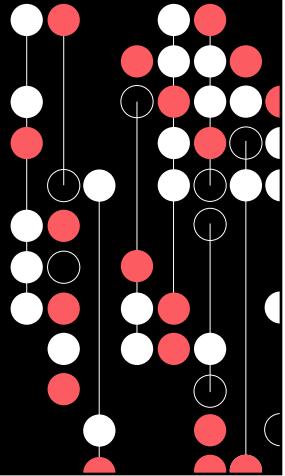


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

ASP.NET Core Minimal API



Session Objectives

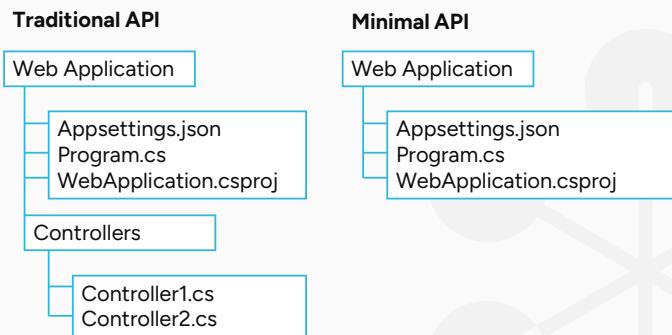
Introduce ASP.NET Core Minimal API, and understand how, when, and why they should be used.

Session Content

- What is ASP.NET Core Minimal API?
- Trad vs Minimal Examples
- When to Use Minimal APIs
- Why Use Minimal APIs
- Adding Endpoints
- Return values
- Route Handlers

What is ASP.NET Core Minimal API?

- Minimal APIs are a new feature in ASP.NET Core for building small HTTP APIs with minimal overhead.
- They provide a simple way to create APIs with fewer files and less boilerplate code.



Trad vs Minimal Examples

Traditional (Controller1.cs)

```
public class ProductsController :  
    ControllerBase  
{  
    [HttpGet]  
    public IActionResult GetAll()  
    {  
        return Ok(new List<string> {  
            "Product1", "Product2" });  
    }  
}
```

Minimal (Program.cs)

```
var builder =  
    WebApplication.CreateBuilder(args);  
  
var app = builder.Build();  
  
app.MapGet("/products",  
    () => new List<string>  
    { "Product1", "Product2" });  
  
app.Run();
```

When to Use Minimal APIs

- Small to medium-sized applications
- Microservices
- Prototyping and quick experiments
- Applications with minimal configuration needs



Why Use Minimal APIs

- Reduced boilerplate code
- Faster development cycle
- Improved performance
- Simplified hosting model



Adding Endpoints

```
app.MapGet("/products", () => GetProducts());  
  
app.MapPost("/products", (Product product) => AddProduct(product));
```



Return Values

```
app.MapGet("/products/{id}", async (int id, ProductContext db) =>
    await db.Products.FindAsync(id)
    is Product product
    ? Results.Ok(product)
    : Results.NotFound());
```

- ASP.NET Core automatically serializes any returned object to JSON and writes the JSON into the body of the response message.
- The response code for this return type is 200 OK, assuming there are no unhandled exceptions.
- If no item matches the requested ID, the method returns a [404 status NotFound](#) error code.
- Unhandled exceptions are translated into 5xx errors.

```
{
  "productID": 3,
  "productName": "Old Knobler",
  "description": "Made with a selected single....",
  "recommendedRetailPrice": 295
}
```

Route Handlers in Minimal API apps

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

app.MapGet("/", () => "This responds to a GET request");
app.MapPost("/", () => " This responds to a POST request ");
app.MapPut("/", () => " This responds to a PUT request ");
app.MapDelete("/", () => " This responds to a DELETE request ");

app.Run();
```



Hands on Lab

Controllers and Actions

Objective:

Your goal is to add an ASP .NET minimal API project to the Microsoft Visual Studio Estate Agent solution that provides CRUD capabilities to a SQL Server database's Sellers table

You will create a microservice for "seller" information. The microservice should be created from an ASP.NET **Minimal API** template and allow a consumer of the service to:

- Retrieve a list of all sellers.
- Retrieve a seller by their id.
- Retrieve a seller by their name.
- Add new sellers.
- Delete existing sellers.
- Update existing sellers.

Summary

Know more about the what, where, why and when of minimal APIs

- Minimal APIs provide a streamlined approach to building APIs
- Ideal for small to medium-sized applications and microservices
- Faster development and improved performance

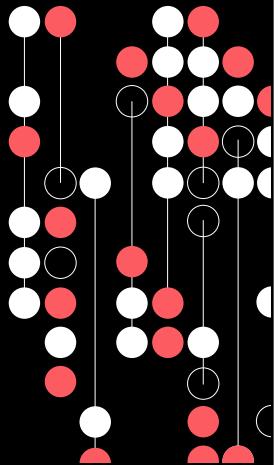


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Microservices



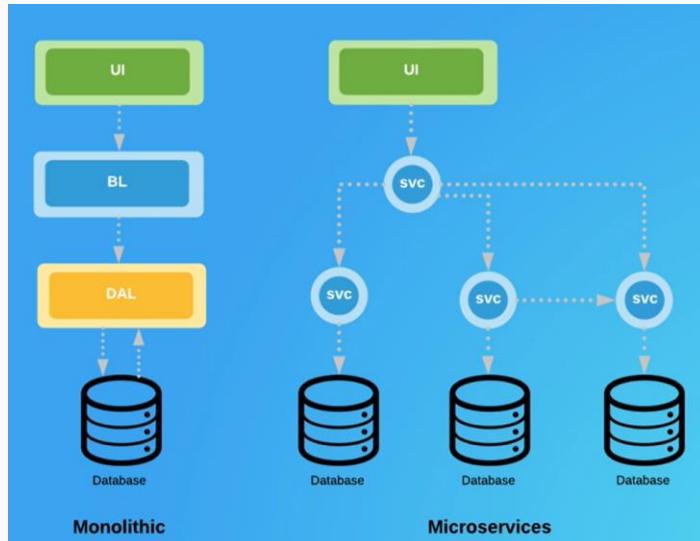
Session Objectives

Introduce the concepts, philosophy and designs of Microservices and understand why they are useful.

Session Content

- Microservice Objectives
- Monolithic vs Microservices
- Internal Structures
- Monolithic Architecture
- Monolithic Physical Deployment
- Microservice Architecture
- Containers
- Containers vs Virtual Machines
- Container Orchestration

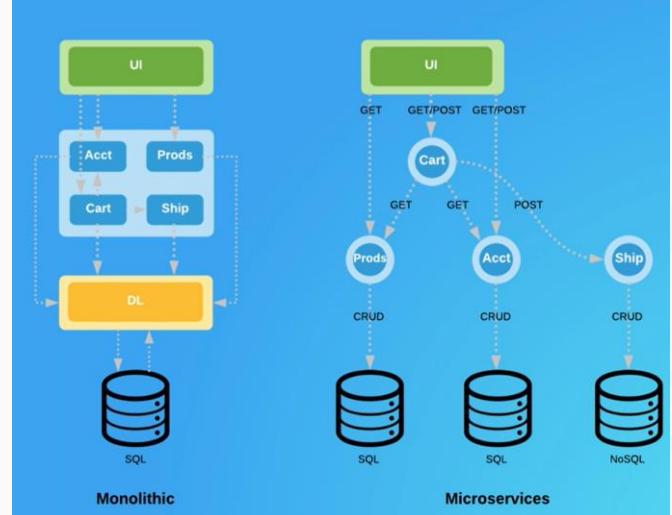
Monolithic vs Microservices



monolithic based applications tend to have a very simple layered structure of components. In this case, a typical three-tiered application exists of a user interface layer, business logic layer and data access layer. Communications passes synchronously from top to bottom, and then back in the reverse. All data resides in a single database. And access to it must be performed through the data access layer. Often, this monolithic design is implemented in a single language and is built and deployed as a single unit.

Meaning that there is often tight coupling between components and/or layers. At the outset of any project adopting this design, implementation testing and deployment is both simple and quick. However, over time, the ability to maintain and add new features, tiers in isolation, deploy and scale the monolithic become increasingly difficult.

Internal Structures



Microservices address the problems of monolithic systems by breaking apart the monolithic into a number of small or loosely coupled microservices. Typically, a microservice models a single distinct business function that when combined together with other microservices provides a complete business application as exposed by the user interface.

Communication between microservices is often performed using APIs where each microservice exposes its functionality as a well-defined interface. Communication is often free to happen between any two microservices, as required by the overall system to work. Each individual microservice can be developed independently of each and every other microservice allowing for polyglot style development and allowing scaling per microservice. Finally, each microservice can be paired with its own flavour of database which is best suited to the microservices specific needs.

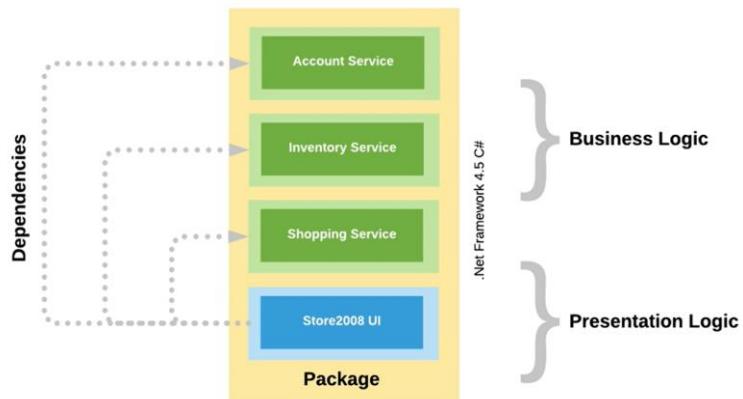
For example, pairing a microservice with a sequel database versus a no sequel database versus a graph-based database. The <https://microservices.io/patterns> website documents additional attributes associated with both the monolithic and microservice based architectures and is highly recommended reading . In

particular, take a look at these two pages

<https://microservices.io/patterns/monolithic.html> and

<https://microservices.io/patterns/microservices.html> website pages . For each style of architecture, the website provides pattern information on context, problem, forces, solution, example, related patterns and real-world implementations.

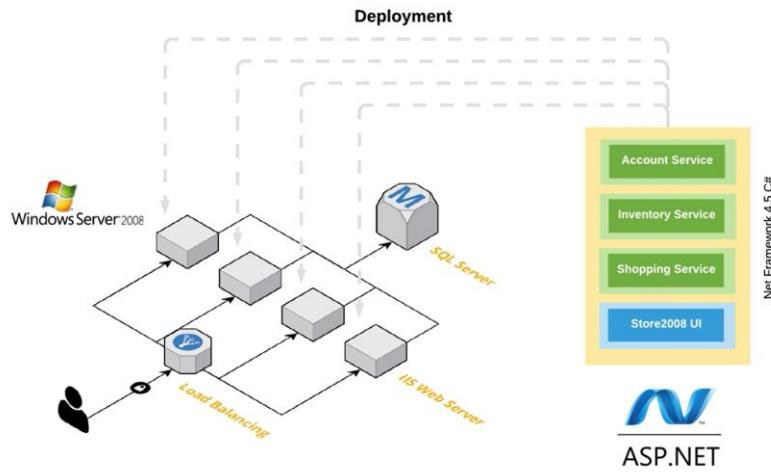
Monolithic Architecture à la 2008



So, what do we mean by monolithic? The slide shows a typical example of an online shop that utilises several components, an account service, inventory service, and shopping service which make up the business logic and back end for the architecture. And then we have our Store2008 user interface component which makes up our presentation logic.

All of these components end up being packaged together and deployed together and that, in essence, is what a monolithic architecture is. So, in this monolithic design, the only component that can actually make use of the business logic components is the presentation logic component. The account service, inventory service, and shopping service components lose the ability to be used elsewhere in the architecture.

Monolithic Physical Deployment

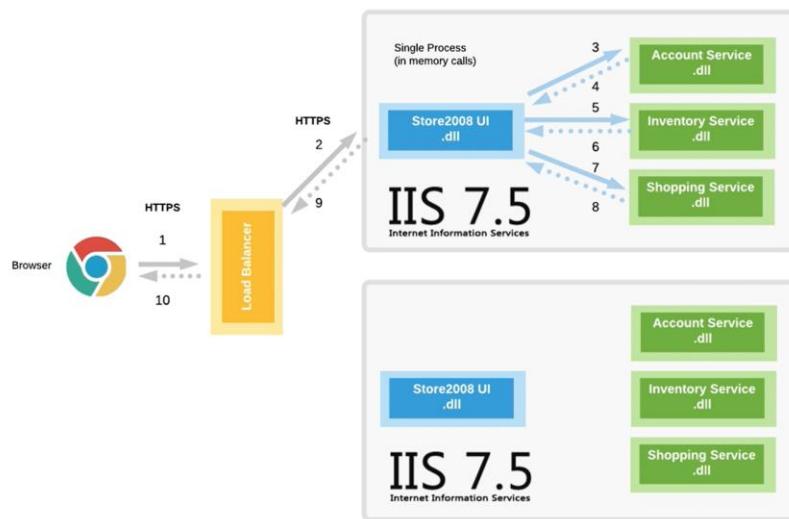


The above diagram represents what the deployment would have looked like for a monolithic architecture 15+ years ago.

You would have typically had a load balancer of some sort and a number of IIS web servers running behind it and then each of those web servers can make connections to a SQL server database and read and write data into the database. The monolithic architecture would have been deployed in its entirety to each of those web servers running behind the load balancer.

So, each and every web server has an identical copy of all of the components.

Call Sequence

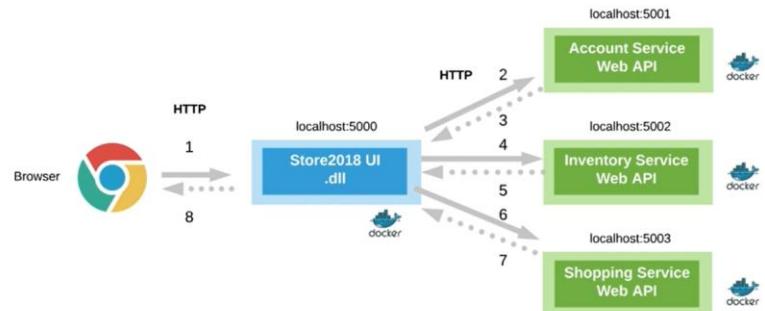


Finally, we can take a look at a diagram that represents the calling for a monolithic architecture. The user's browser would make HTTPS calls to a load balancer. The load balancer would have to load balance the calls across a number of IIS web servers.

The deployment of the monolithic set up would be made up of 4 dlls, one for each of the components. The store2008 user interface would be its own .NET assembly, as would of each of the individual services. The monolithic design here is deployed in an identical manner to each and every web server that sits behind the load balancer. And one of the other key points in terms of how the calling is made between the components is that the presentation UI is actually making in-memory calls to each of the service components.

All of these components are effectively running in a single process within IIS. Again, a key constraint of a monolithic architecture is that you lose the ability to expose services to a wider range of other services. In the example, the account service, inventory service, and shopping service can only be called currently by the Store2008 user interface component.

Microservice Architecture (Post 2018)



Can be deployed in a scalable manner via Kubernetes

Containers

Containers are a form of virtualization that:

- Operate at system level
- Allow applications, dependencies, libraries, and configurations to be packaged into a single unit.
- Ensure applications can run reliably and consistently across different environments

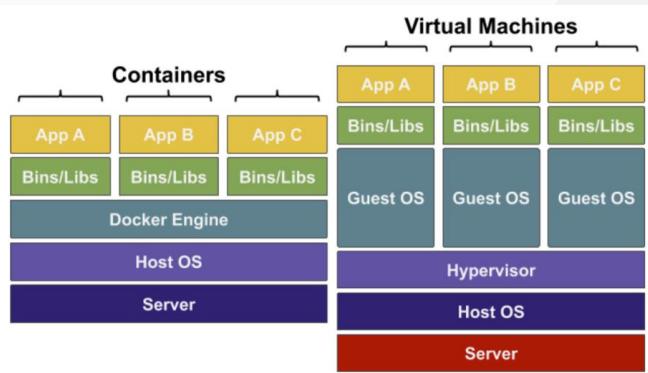
Containers are:

- Lightweight
- Efficient
- Portable
- Crucial components in modern IT and DevOps practices.

Containers vs Virtual Machines (VM)

Containers and VMs are both virtualisation tools

- Run on the same kernel as host
 - Resources are shared
 - Require fewer resources
 - Designed to perform 1 task only
- Emulate an entire computer
 - Require a hypervisor
 - Resources can't be shared
 - Require a lot of resources to run



Containers and virtual machines are both methods of *virtualisation*, but they have a number of key differences. The diagram above illustrates the structural differences between the two.

Virtual Machines

- Emulate an entire computer to support a full operating system
- Require a hypervisor to allocate and reserve compute resources (CPU, RAM, storage, etc.) - these resources cannot be shared between VMs
- Require a lot of resources to run

Due to their nature, virtual machines are extremely versatile but require far more resources. This makes them slow to start up. In a project environment, efficiency and effectiveness are key.

Containers

- Run on the same kernel (the operating system's core process) as the host machine, meaning it has direct access to the compute resources it needs to use (i.e. no need for a hypervisor)
- Resources are shared between containers
- Require far fewer resources to run
- Are typically designed to perform one task only

Unlike virtual machines, containers are extremely lightweight, making them very

quick to start up. They are less versatile than virtual machines in their abilities but are perfectly suited when you only need to perform one task (e.g. host a web application).

Container Orchestration

Containers need to be managed so as to provide:

- Scalability
- Resilience
- Portability
- Efficiency and High availability
- Ease of Management
- CI/CD

Tools that can do this include:

- Kubernetes
- Nomad
- Amazon ECS (Elastic Container Service)
- Azure Service Fabric
- Docker Swarm

Hands on Lab

Controllers and Actions

Objective:

Take the services created in the previous labs and edit them so they each use their own single table database and to address the referential integrity issues the changes create.

In the current setup all the microservices use the same database which is configured to "cascade delete" any dependencies. In our case this means when a property is removed from the properties table the database ensures any associated bookings for that property are automatically deleted from the "bookings" table. In the "new world" of microservices we are encouraged to develop services that each use their own database such that the databases only host a minimal number of tables (typically just one). This means we, as developers, need to worry about the referential integrity of our data rather than letting the databases do it for us.

Summary

Understand the philosophy and structure of Microservices

- Microservice Objectives
- Monolithic vs Microservices
- Internal Structures
- Monolithic Architecture
- Monolithic Physical Deployment
- Microservice Architecture
- Containers
- Containers vs Virtual Machines
- Container Orchestration



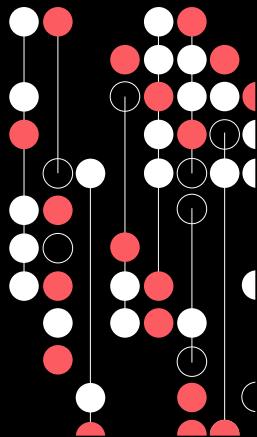


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Introduction to Containerisation using Docker Desktop



Session Objectives

Understand how to setup Docker containers and deploy .NET Microservices to them.

Session Content

- Introducing Docker
- Introducing Docker Security
- Images
- Image Layers
- Registry
- Image Properties and Commands
- Containers
- DockerFiles
- Multi-Stage Builds
- DockerIgnore
- Networking
- Volumes
- Visual Studio Integration

Introducing Docker



- Widely used in Dev-Ops, for its lightweight and rapid deployment
- Can be used on multiple platforms in use of environment standardisation
- Speeds up cumbersome task of ensuring all team members can get access to it
- Designed to work on any operating system on any platform
 - Helps in the cloud due to the rising issue of interoperability
 - Promotes portability with its ability to bundle dependencies within the container
- Is also used in workflows such as during development
 - Integrates CI/CD methodologies

Docker is used widely in Dev-Ops, for its lightweight and rapid deployment. When projects are spread amongst a team, the convenience of having an application that can be used on multiple platforms in use of environment standardisation, speeds up the cumbersome task of ensuring all members can get access to it. From the standpoint of a project lead, any method that can cut down menial tasks and add efficiency to a project is a major plus. The flexibility that containerisation offers both on-premise and in the cloud is a huge advantage. Docker containers abstract at the OS level rather than at application level, meaning that Docker is intentionally designed to work on any operating system on any platform. This especially helps in the cloud due to the rising issue of interoperability, which Docker tackles with its platform-agnostic containers. This is then paired with the portability Docker offers with its ability to bundle dependencies within the container - altogether resulting in a technology and its containers being highly flexible!

Docker is not limited to a single use of deployment and is used in workflows such as during development. It helps set up the environment and save time for new developers to start projects in their preferred programming language.

Docker is used in different stages and gives developers a chance to try new technologies. It integrates the CI/CD methodologies and allows collaboration between the team members to share docker images.

Docker can be helpful in times of disaster recovery. The lost data can be retrieved at a later point if there are any serious issues. For example, if there is a hardware failure, data can be replicated from Docker onto new hardware

quickly to keep workflow continuous.

Introducing Docker - Security

- There are also security elements to Docker that VMs and bare-metal don't provide without scrutinous effort:
- The Docker images are transparent and easily readable.
- As a micro-service, you can link certain security problems to specific locations.
- By using containers, the security focus narrows down to the host system. This results in a reduced attack surface to defend.
- Docker also makes for easy updates, pulling latest versions of images and applying patches in quick response to known vulnerabilities.

- The Docker images are transparent and easily readable which aids understanding of what potential security risks could be present.
- As a micro-service, you can link certain security problems to specific locations, thus making it easier to find and resolve vulnerabilities.
- By using containers, the security focus narrows down to the host system, the Docker daemon (which is considerably smaller than a full virtual operating system), along with the application inside the container. This results in a reduced attack surface to defend.
- Docker also makes for easy updates, pulling latest versions of images and applying patches in quick response to known vulnerabilities. This can be done with minimal disruption to end users.

Mini Lab:

Install Docker Desktop and Run Hello World container

Install from here:

[Docker Desktop: The #1 Containerization Tool for Developers | Docker](#)

Note: For Windows, ensure you enable HyperV for Windows.

Open PowerShell or a Command prompt

Enter:

docker run --rm hello-world

The rm flag removes the container when it exits or when the daemon exits, whichever happens first.

Try something more ambitious, you can run an Ubuntu container with:

docker run -it ubuntu bash

Images

- Docker images are the files used to create containers.
 - They are an isolated snapshot of an environment
 - Are typically used to package applications.
 - Are read-only
 - Exist in a layered structure
 - Are the heart of containerization technology
- To create a container of an environment or application on a foreign machine we can:
 - Provide an image, which:
 - Eliminates the need to manually create the environment in the container
 - And allows the container to be spun up instantly



Image Layers

- Images are read-only
- Add new layers to customize them thus creating a new image
- Benefits of Layering:
 - Saves space. If multiple images on the disk use the same layers, those layers will be saved on the disk once.
 - Saves time. It eliminates the need for repeating installation or configuration steps for each image.

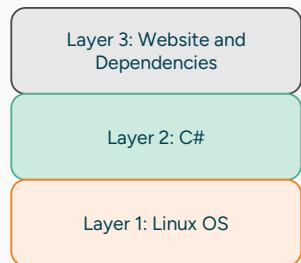
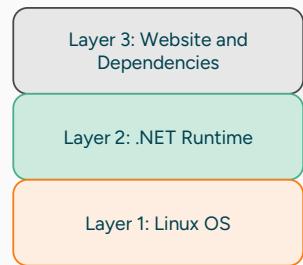


Image Layers

- Images are read-only
- Add new layers to customize them thus creating a new image
- Benefits of Layering:
 - Saves space. If multiple images on the disk use the same layers, those layers will be saved on the disk once.
 - Saves time. It eliminates the need for repeating installation or configuration steps for each image.



Registry - 1

- Images are typically stored in remote registries, so they are more accessible.
- Docker Hub is the Official Registry
 - Used for searching and pulling images
- Microsoft have the Microsoft Artifact Registry for all things .NET
 - <https://mcr.microsoft.com/en-us/catalog?alphaSort=asc&alphaSortKey=Name>
 - No need to sign in
 - `docker pull mcr.microsoft.com/dotnet/runtime:8.0`
 - `docker pull mcr.microsoft.com/dotnet/sdk:8.0`

[Microsoft Artifact Registry](https://mcr.microsoft.com/en-us/catalog?alphaSort=asc&alphaSortKey=Name): <https://mcr.microsoft.com/en-us/catalog?alphaSort=asc&alphaSortKey=Name>

Current (as of May 2024) images:

Image	Comments
mcr.microsoft.com/dotnet/aspnet:8.0	ASP.NET Core, with runtime only and ASP.NET Core optimizations, on Linux and Windows (multi-arch)
mcr.microsoft.com/dotnet/sdk:8.0	.NET 8, with SDKs included, on Linux and Windows (multi-arch)

Registry - 2

- When you run a container with an image that you didn't build (mcr.microsoft.com/dotnet/aspnet:8.0 for instance) the image will be downloaded from the registry first.
- You can upload and pull your own images to and from Docker Hub
- You can create your own registries
 - Useful when you don't want your images to be publicly accessible

Image Properties

- The full reference to an application's image follows the format:
- [HOST] / [AUTHOR] / [APPLICATION] : [TAG]**
- [HOST] is the remote registry where the image is stored. It defaults to Dockerhub.
- [AUTHOR] is the uploader of the image.
- [APPLICATION] is the name of the application being containerised.
- [TAG] is the version of the application. If no tag is specified, it defaults to latest.
- The Docker CLI assigns a few properties to images:

	Name	Tag	Status	Created	Size	Actions
<input type="checkbox"/>	store2022microservice	dev	In use	5 seconds ago	216.65 MB	
<input type="checkbox"/>	accountservice	dev	In use	5 seconds ago	216.65 MB	
<input type="checkbox"/>	inventorystorage	dev	In use	5 seconds ago	216.65 MB	
<input type="checkbox"/>	shoppingservice	dev	In use	5 seconds ago	216.65 MB	

Property	Description
Repository	The application. This is the name of the repository in the remote registry (typically Docker Hub) where this image is stored.
Tag	This property is used for versioning. You can reference or tag images using the format [IMAGE]:[TAG]
Image ID	This is a unique identifier for the image. This is a reliable way of referencing images.
Created	When the image was created.
Size	Size of image on disk.

Docker Image Commands

Command	Usage	Description
docker search	docker search [IMAGE]:[TAG]	Search for images
docker pull	docker pull [IMAGE]:[TAG]	Download an image
docker push	docker push [USERNAME]/[IMAGE]:[TAG]	Uploads an image to registry (must be logged in)
docker tag	docker tag [OLD_IMAGE]:[OLD_TAG] [USERNAME]/[NEW_IMAGE]:[NEW_TAG]	Tag an image or rename it
docker rmi	docker rmi [IMAGE]/[TAG]	Delete an image from local disk

Mini Lab: Register for Docker

Docker MINI LAB 1: Register for Docker



Containers

- Lightweight virtual environments that are used to package up code with its necessary dependencies.
- Providing our applications with their own environment allows them to be recreated across any machine with Docker on it.
- The Docker CLI (and Docker Desktop) provides us with the necessary tools to deploy and manage containers.
- containers have several properties that provide us with useful information about them:

Name	Image	Status	Port(s)	CPU (%)	Last started	Actions
deckermoose70807222361497903	store0222microservice	Running	4/4	2.19%	2 hours ago	[Stop] [Restart]
store0222Microservice	store0222microservice.dev	Running		0.52%	2 hours ago	[Stop] [Restart]
InventoryService	shop00service.dev	Running		0.53%	2 hours ago	[Stop] [Restart]
70347a7b3991	shop00service.dev	Running		0.55%	2 hours ago	[Stop] [Restart]
AccountService	accountservice.dev	Running		0.59%	2 hours ago	[Stop] [Restart]
19a477401144	accountservice.dev	Running		0.54%	2 hours ago	[Stop] [Restart]
InventoryService	inventoryservice.dev	Running		0.54%	2 hours ago	[Stop] [Restart]
3a0709990006	inventoryservice.dev	Running		0.54%	2 hours ago	[Stop] [Restart]

Docker containers have several properties that provide us with useful information about them.

Property	Description
Container ID	The unique identifier for the container.
Image	The Docker image used for the container
Command	The main process running inside the container.
Created	The amount of time since the container was created.
Status	Whether the container is running or not, and how long.
Ports	The exposed and published ports of the container.
Names	The name of the container.

Containers Use Case

Consider a CI/CD pipeline for an application. As new code is integrated, it will be migrated across multiple environments, such as development, testing and production environments.

- It's highly likely that these environments will differ in their configurations
 - potentially causing the application to behave erratically.
- For example:
 - A developer may have a specific software dependency installed on their development environment, as well as environment variables assigned that the application utilises. If these specific dependencies aren't replicated across the pipeline, the application will not run correctly.
 - Containerising the application with its dependencies allows us to migrate the environment along with the application, meaning it will perform the exact same way on any machine running Docker.

Starting up a Container

Docker Run is the command to start up Docker containers:

`docker run [OPTIONS] [IMAGE]:[TAG]`

e.g.

`docker run -d -p 5000:80 -name accserve accserveimg`

docker run : Starts a container

-name accserve : specifies the name being given to the container. If no name is specified a random name is generated

accserveimg : The name of the image being used

-p : allows you to specify ports. In the example the container's port 80 is mapped to the host machine's port 5000.

-d : Runs the container in detached mode (logs are not output to the terminal).

Docker ps : Returns a table of containers

Here's a list of some commonly used options:

Option	Format	Description
--detached or -d	<code>docker run -d</code>	Runs the container in detached mode. The container logs are not outputted to the terminal
--publish or -p	<code>docker run -p [HOST PORT]:[CONTAINER PORT]</code>	Publishes a port. Maps a port inside of the container to a port on the host so that it is accessible from the host's IP address.
--name	<code>docker run --name [NAME]</code>	Names the container. If no name is set, the container is given a randomly generated name.
--env or -e	<code>docker run -e [VARIABLE_NAME]=[VARIABLE_VALUE]</code>	Sets an environment variable inside the container

Mini Lab: Setting up a container

Docker MINI LAB 2: Run and Test a Container



DockerFiles

- A Dockerfile is a set of instructions executed in a step-by-step fashion to build Docker images.
 - Text based
 - Contains all necessary commands that would be used on a command line to create a container image.
- When building an image for Docker using the docker build command:
 - The Dockerfile is where the instructions come from.
 - A Dockerfile is a text file called DockerFile with no file extension.
 - We just need a text editor.
- Each instruction in a Dockerfile creates an intermediate image that is saved (the layers of the image).
 - For example: if there are four instructions in a Dockerfile and your build fails on the fourth, reattempting the build will restart the build process from step 4.
- Use docker build to build the image

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
EXPOSE 8091

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["AccountService/AccountService.csproj",
      "AccountService/"]
RUN dotnet restore "./AccountService/AccountService.csproj"
COPY . .
WORKDIR "/src/AccountService"
RUN pwd
RUN ls -la
RUN dotnet build "./AccountService.csproj" -c
$BUILD_CONFIGURATION -o /app/build
/p:UseAppHost=false

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./AccountService.csproj" -c
$BUILD_CONFIGURATION -o /app/publish
/p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "AccountService.dll"]
```

200

There are a couple of options available when building an image.

Option	Usage	Description
-f	docker build -f [PATH_TO_DOCKERFILE] [CONTEXT]	Execute a Dockerfile from a different directory.
-t	docker build -t [USER]/[IMAGE]:[TAG]	Tag (name) the image being created.

Multi-Stage Builds - 1

- Multi-stage builds allow you to separate the build stages of your images into different containers.
- Beneficial for keeping your image and container sizes down
 - Build tools and source code can be used to compile code in temporary intermediate containers and then deleted once you no longer need them.
- Commonly used for compiled languages, such as C#, which generate dll's (and exe's)
- For consistency of environments, it is beneficial to perform the build stage in a container based on a build tool image.
 - For C# microservices, you will want to compile it in a container based on an ASP.NET Core or .NET sdk image
 - see https://hub.docker.com/_/microsoft-dotnet-aspnet/ and https://hub.docker.com/_/microsoft-dotnet-sdk/
- You can then set the compiled dll as your entrypoint.

309

Here's a list of some commonly used options:

Option	Format	Description
--detached or -d	docker run -d	Runs the container in detached mode. The container logs are not outputted to the terminal
--publish or -p	docker run -p [HOST PORT]:[CONTAINER PORT]	Publishes a port. Maps a port inside of the container to a port on the host so that it is accessible from the host's IP address.
--name	docker run --name [NAME]	Names the container. If no name is set, the container is given a randomly generated name.
--env or -e	docker run -e [VARIABLE_NAME]=[VARIABLE_VALUE]	Sets an environment variable inside the container

Multi-stage Builds - 2

Multi-stage builds allow us to split our build and execution stages into two (or more) separate containers: one with the build tool installed, the other with the application's runtime installed.

- The steps are:
1. Perform the compiling in the build tool image
 2. Copy the build artefact from the build stage into the runtime image
 3. Delete everything associated with the build stage

This has the benefit of:

- Keeping our build environments consistent
- Keeping our application's image and container size down
- Speeds up deployment of containers

310

For a couple of more detailed discussions on the topic of multi-stage builds and keeping the image sizes down see:

[Optimizing your .NET Core Docker image size with multi-stage builds | by Chris Lewis | Medium](https://medium.com/@chrislewisdev/optimizing-your-net-core-docker-image-size-with-multi-stage-builds-778c577121d): <https://medium.com/@chrislewisdev/optimizing-your-net-core-docker-image-size-with-multi-stage-builds-778c577121d>

And

[Optimizing ASP.NET Core Docker Image sizes - Scott Hanselman's Blog](https://www.hanselman.com/blog/optimizing-aspnet-core-docker-image-sizes):
<https://www.hanselman.com/blog/optimizing-aspnet-core-docker-image-sizes>

DockerFile Instructions

Instruction	Description
FROM	Define the base image.
RUN	Execute a command on an intermediate container.
CMD	Define the main process of the container OR define default variables for ENTRYPOINT.
ENTRYPOINT	Define the main process for the container to run on start-up.
LABEL	Set metadata for the image.
EXPOSE	Selects the port that the container listens on.
ENV	Set an environment variable.
COPY	Copy files into image.
ADD	Copy files into image (source can be URL).
WORKDIR	Sets the working directory for RUN, CMD, ENTRYPOINT, COPY and ADD instructions.
ARG	Defines variables for use during build-time.
VOLUME	Creates a mount point with the specified name to save data externally.

Mini Lab: Creating and using a DockerFile

Docker MINI LAB 3: Creating and Using
a Docker File



Docker Compose

A tool for Docker that allows you to define and run multiple Docker containers with a single command

- Uses a single YAML configuration file where we can specify the deployment completely.
 - docker-compose.yml
- Excellent tool for defining and creating collections of containers.
- Great for spinning up microservice applications.
- Invoked via
 - docker-compose build – Rebuilds app from source
 - docker-compose up – Picks up on changes to a services configuration and/or image and applies them to the containers by stopping and recreating them whilst preserving any mounted volumes

Docker Compose CLI

Command & Usage	Option	Description
docker-compose up [OPTIONS]		Start up all the services declared in the configuration
	-d, --detach	Run the containers in the background (doesn't hang the terminal)
	--build	Build images before starting containers
	--scale [SERVICE]=[NUM]	Can specify the number ([NUM]) of containers for a service. Overrides the scale setting in the Compose file.
docker-compose down [OPTIONS]		Stops the services created up.
	--rmi [TYPE]	Removes images. [TYPE] has two options, all and local. all removes all images, and local removes images built by Docker Compose.
docker-compose build [OPTIONS]		Builds the images defined in the configuration without running any containers.
	--parallel	Builds all the images at the same time, useful for saving time if the images do not depend on each other. -
docker-compose ps [OPTIONS]		Returns all the running containers started by the Docker Compose configuration
	-a, --all	Show all containers, including stopped ones
docker-compose exec [OPTIONS] [SERVICE] [COMMAND]		Execute a command in a running container
	-d, --detach	Run the command in the background
	--index=index	Run the command in a specific container. The default is 1.
	-e, --env	Set environment variables
docker-compose logs [SERVICE...]		Display the log outputs from services that are running. If no services specified, then all logs are displayed
	-f, --follow	Displays the log outputs in real time
docker-compose push		Push the images used for the services to a registry

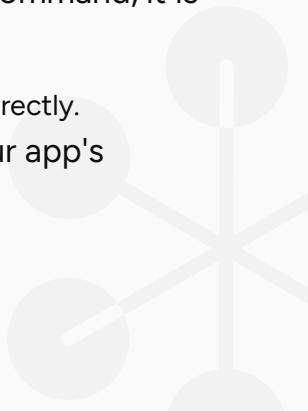
The `.dockerignore` file

Dockerignore is a file that can be placed at the root of the context, called `.dockerignore`.

- It contains a list of files and/or directories that we want Docker to ignore during the image building process (and also when pushing projects to GitHub).
- During the image building process, a core step is adding files from a source/context into the image.
- It is often the case that we do not need to add every file into the image. For instance:
 - Some files in the context may contain credentials and aren't needed in the image and copying them into the image may create a security risk.
 - When there are large files in the context that aren't going to be used. Adding them into the image would be a waste of time and cause the image size to be larger than it needs

Networking

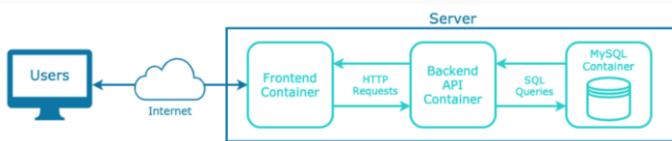
- When you run a container with the docker run command, it is assigned:
 - To a network called default.
 - An IP address which allows it to be communicated with directly.
- This is fine for monolithic applications, where all our app's functionality is located in the one container.



Networking - Possible Issue?

Consider a web application that consists of the following:

- A frontend server that communicates directly with users over the internet to serve them HTML pages
- A backend server that handles the app's internal logic
- the frontend communicates with the backend via API calls
- A database client that persists user information - it only communicates with the backend container



If each container is assigned an indeterminate IP address by the Docker Engine, how can we hard-code the address locations for our containers?

- Thankfully, Docker provides other types of networking solutions that allow us to circumvent this problem.

Network Types

Docker provides a number of different network drivers, each designed for different networking purposes. The most common ones include:

- Bridge (Default)
- Overlay
- Host

This course only deals with Bridge Networks

318

Bridge Network (Default)

The Bridge network is the default network driver for Docker containers. When you run a container without specifying a network, it is connected to the Bridge network. This network is used for communication between containers on the same Docker host.

Overlay Network

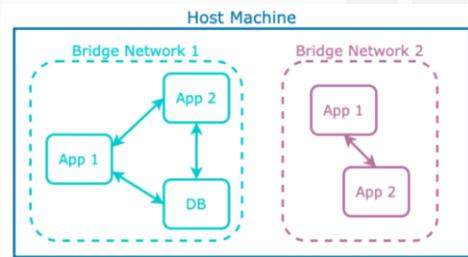
The Overlay network is used to enable communication between Docker containers across different Docker hosts. It is commonly used in Docker Swarm and Kubernetes to manage a cluster of Docker engines. Overlay networks allow services to communicate with each other, even if they are running on different physical hosts.

Host Network

The Host network mode removes the network isolation between the Docker container and the Docker host. The container shares the host's network stack, meaning that it has direct access to the host's networking.

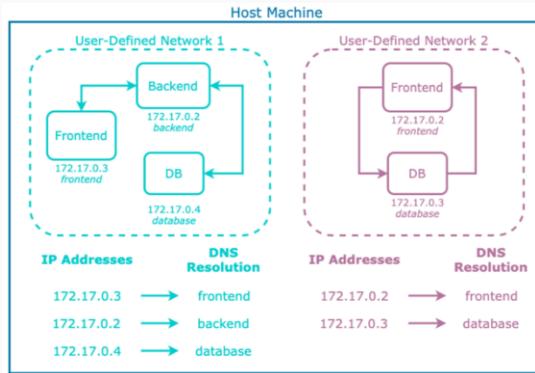
Bridge Networks

- Default network type
- Used for connecting multiple containers on a single host.
 - The default network will assign an IP address to containers, which is unhelpful for allowing our applications to talk to one another as we can't easily anticipate what that IP will be.
- Allows containers to be accessed via a private IP address.
- Can create user-defined bridge networks which:
 - Allow you to assign DNS names to your containers.
 - Also allows you to create multiple, self-contained networks.



User Defined Bridge Networks

- IP addresses and network types can be defined as part of a docker-compose file



320

An example docker-compose.yml file that defines the networking for a set of three microservices that service a ASPNET UI that is also being hosted in a docker container:

```

services:
  store2022microservice:
    image: ${DOCKER_REGISTRY-}store2022microservice
    environment:
      ACCOUNT_SERVICE_API_BASE: http://172.19.10.101:5001/api
      INVENTORY_SERVICE_API_BASE: http://172.19.10.102:5002/api
      SHOPPING_SERVICE_API_BASE: http://172.19.10.103:5003/api
    build:
      context: .
      dockerfile: Store2022Microservice/Dockerfile
    ports:
    - "3000:3000"
    - "3001:3001"
    networks:
      Store2022Microservicenetwork:
        ipv4_address: 172.19.10.100

accountservice:
  image: ${DOCKER_REGISTRY-}accountservice
  build:
    context: .
    dockerfile: AccountService/Dockerfile
  
```

```
ports:
- "5001:5001"
networks:
  Store2022Microservicenetwork:
    ipv4_address: 172.19.10.101
inventoryservice:
  image: ${DOCKER_REGISTRY-}inventoryservice
  build:
    context: .
    dockerfile: InventoryService/Dockerfile
  ports:
- "5002:5002"
  networks:
    Store2022Microservicenetwork:
      ipv4_address: 172.19.10.102

shoppingservice:
  image: ${DOCKER_REGISTRY-}shoppingservice
  build:
    context: .
    dockerfile: ShoppingService/Dockerfile
  ports:
- "5003:5003"
  networks:
    Store2022Microservicenetwork:
      ipv4_address: 172.19.10.103

networks:
  Store2022Microservicenetwork:
    driver: bridge
    ipam:
      driver: default
      config:
        - subnet: 172.19.0.0/16
```



User Defined Bridge Networks DEMO

Launch demo app and test out the following urls:

<https://localhost:3001/home/index>

<http://localhost:5001/api/consumers>

<http://localhost:5002/api/products>

<http://localhost:5003/api/cart/5>

200

Copy of the Store2022Microservice project's DOCKERFILE

#See <https://aka.ms/customizecontainer> to learn how to customize your debug container and how Visual Studio uses this Dockerfile to build your images for faster debugging.

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
USER app
WORKDIR /app
ENV DOTNET_URLS=http://+:3000
ENV DOTNET_URLS=https://+:3001
EXPOSE 3000
EXPOSE 3001

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
ARG BUILD_CONFIGURATION=Release
WORKDIR /src
COPY ["Store2022Microservice/Store2022Microservice.csproj", "Store2022Microservice/"]
RUN dotnet restore "./Store2022Microservice/Store2022Microservice.csproj"
COPY . ./src/Store2022Microservice"
RUN pwd
RUN ls -la
RUN dotnet build "./Store2022Microservice.csproj" -c $BUILD_CONFIGURATION -o /app/build

FROM build AS publish
ARG BUILD_CONFIGURATION=Release
RUN dotnet publish "./Store2022Microservice.csproj" -c $BUILD_CONFIGURATION -o /app/publish
/p:UseAppHost=false

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Store2022Microservice.dll"]
```

Volumes

- Containers are designed to be spun up and ripped down quickly and easily
 - Consequently, they do not persist data automatically.
 - What if database and its data for web app is hosted in a container?
 - If the database container is lost its stored data will also be lost!
- Volumes are a way of persisting data created and used by a container.
 - Volumes are managed by Docker and do not rely on the host's file system.
 - Volumes can be used by multiple containers, enabling them to share data between them.
 - Volumes can be managed using the Docker CLI.
 - Using volume drivers, volumes can be stored on remote hosts
 - Volumes can be shared between multiple containers



Docker and Visual Studio

Visual Studio makes it easy(ish) to integrate ASP.NET API microservices with Docker!

Docker Support and Tools Integration: Provides built-in Docker support

Containerized Development: Develop and run your ASP.NET API microservices inside Docker containers.

Debugging Inside Containers: Debug your ASP.NET API microservices running inside Docker containers.

Docker Compose for Multi-Container Apps: Supports Docker Compose

CI/CD Pipeline Integration: Integrate with CI/CD pipelines to automate the build, test, and deployment of your Dockerized ASP.NET API microservices.

FIND OUT MORE IN THE LAB!

325

Docker Support and Tools Integration: Visual Studio provides built-in Docker support, allowing you to easily add Docker support to your ASP.NET API projects. This includes generating Dockerfiles, managing Docker Compose configurations, and utilizing Docker CLI commands directly from the IDE.

Containerized Development: Visual Studio enables you to develop and run your ASP.NET API microservices inside Docker containers. This ensures that your development environment closely matches your production environment, reducing the chances of environment-related issues.

Debugging Inside Containers: With Visual Studio, you can debug your ASP.NET API microservices running inside Docker containers. This includes setting breakpoints, stepping through code, inspecting variables, and more, just as you would with a locally running application.

Docker Compose for Multi-Container Apps: Visual Studio supports

Docker Compose, allowing you to define and run multi-container applications. You can orchestrate multiple ASP.NET API microservices and their dependencies (like databases and message queues) using a single Docker Compose file, simplifying development and testing.

CI/CD Pipeline Integration: Visual Studio can be integrated with CI/CD pipelines to automate the build, test, and deployment of your Dockerized ASP.NET API

microservices. By leveraging Azure DevOps or GitHub Actions, you can create pipelines that automatically build Docker images, run tests inside containers, and deploy the microservices to various environments.

Docker Support and Tools Integration: Visual Studio provides built-in Docker support, allowing you to easily add Docker support to your ASP.NET API projects.
Containerized Development: Visual Studio enables you to develop and run your ASP.NET API microservices inside Docker containers.

- Ensures that your development environment closely matches your production environment
- Reducing the chances of environment-related issues.

Debugging Inside Containers: With Visual Studio, you can debug your ASP.NET API microservices running inside Docker containers. This includes setting breakpoints, stepping through code, inspecting variables, and more, just as you would with a locally running application.

Docker Compose for Multi-Container Apps: Visual Studio supports Docker Compose, allowing you to define and run multi-container applications.

- Can orchestrate multiple ASP.NET API microservices and their dependencies (like databases and message queues) using a single Docker Compose file

CI/CD Pipeline Integration: Visual Studio can be integrated with CI/CD pipelines (Using GitHub or Azure DevOps) to automate the build, test, and deployment of your Dockerized ASP.NET API microservices.

Hands on Lab

Controllers and Actions

Objective:

Your goal is to learn how to containerize projects using Visual Studio and Docker Desktop

You will reengineer the Estate Agent services so that they each run in their own Docker container and in addition, each service will make use of a dedicated SQL server service that hosted in their own Docker containers.

Summary

Now Know:

- Docker
- Images
- Image Layers
- Registry
- Image Properties and Commands
- Containers
- DockerFiles
- Multi-Stage Builds
- DockerIgnore
- Networking
- Volumes
- Visual Studio Integration



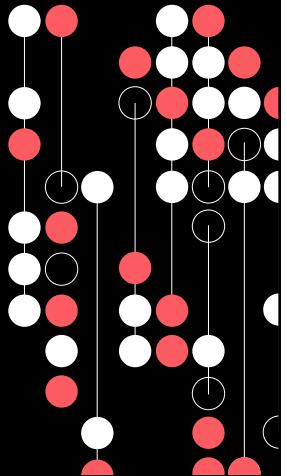


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Event Bus



Session Objectives

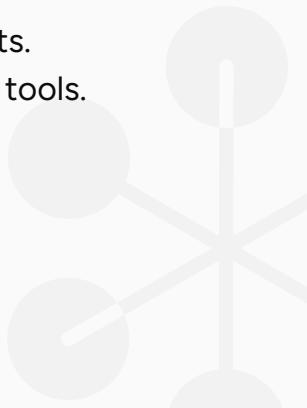
Introduce ASP.NET Core Minimal API, and understand how, when, and why they should be used.

Session Content

- Introduction to the Event Bus Pattern
- What is the Event Bus Pattern?
- Where is the Event Bus Pattern Used?
- Why Use the Event Bus Pattern?
- Implementation in ASP.NET Minimal API with CAP and RabbitMQ

Introduction to the Event Bus Pattern

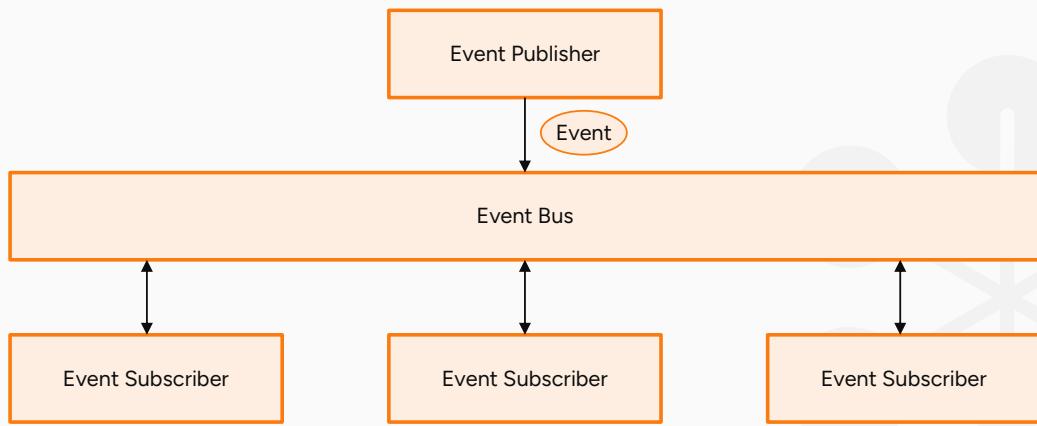
- Event-driven architecture and its importance.
- The role of the Event Bus in decoupling components.
- Overview of CAP (A .NET library) and RabbitMQ as tools.



What is the Event Bus Pattern?

- A design pattern that facilitates event-based communication between different parts of an application.
- Core components: Event Producers (Publishers), Event Consumers (Subscribers), and the Event Bus (see next slide).
- Ensures loose coupling by using an intermediary (the Event Bus) for communication.

UML for the Event Bus Pattern



Event Publisher: A component that is responsible for publishing an Event to the Event Bus.

Event Bus: After the Event Bus receives an Event from Event Publisher, the Bus notifies all the subscribers

Event Subscriber: The Event Subscriber receives an Event, and it will respond accordingly.

Why Use the Event Bus Pattern?

- **Loose Coupling:** Components don't need direct knowledge of each other.
- **Scalability:** Easily add new producers or consumers without major changes.
- **Flexibility:** Modify event flows without disrupting existing systems.
- **Asynchronous Processing:** Publishers and subscribers operate independently, allowing tasks to be processed without waiting for a response.

Cons to using an Event Bus

- **Eventual Consistency:**

- May lead to temporary data inconsistencies
- Updates to different services may not be immediately reflected across the system

- **Complexity in Error Handling:**

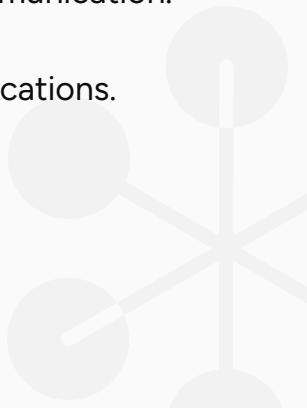
- Increases the complexity of ensuring data integrity and system reliability
- If an event fails to be processed, it can lead to inconsistencies or require complex retry and compensation logic

- **Complexity in Data Management:**

- Managing data consistency and integrity across services can be challenging

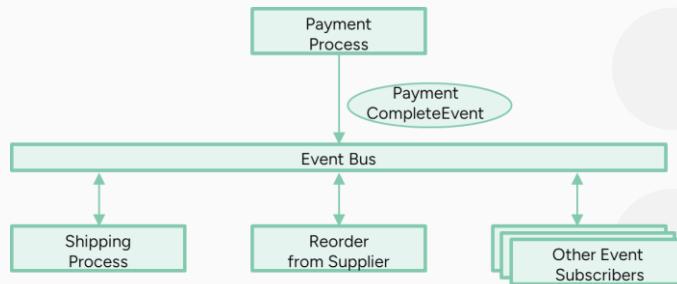
Where is the Event Bus Pattern Used?

- Microservices architectures for asynchronous communication.
- Scaling and Distributing Workloads.
- Decoupling business logic layers in enterprise applications.



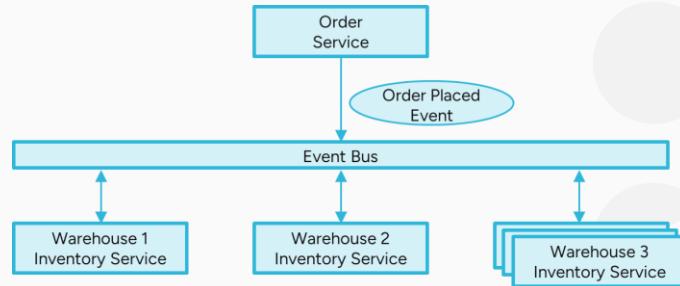
Handling Asynchronous Processes- Example

A payment processing service might publish an event after a payment is completed. Downstream services like shipping or billing can process this event asynchronously without holding up the payment process.



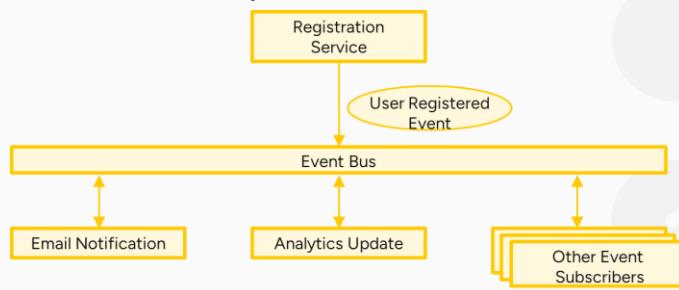
Scaling and Distributing Workloads.

An e-commerce system, an order service can publish an event when a new order is placed, and multiple inventory services can subscribe to the event to update stock levels across different warehouses.

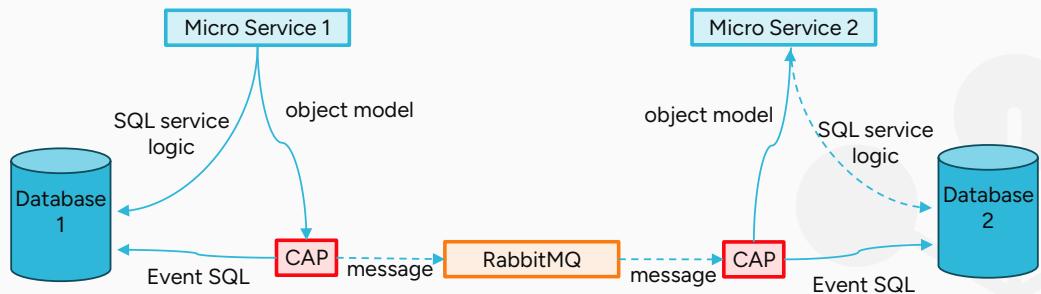


Decoupling business logic layers in enterprise applications

If a user registration service needs to trigger an email notification and an analytics update, the use of an Event Bus allows these services to subscribe to the registration event without the registration service needing to know about the specific details of these downstream services.



Implementation Overview: ASP.NET Minimal API



- **CAP**: a .NET library for distributed transactions
- **RabbitMQ**: a message broker

Setting up the Projects

For both publisher and subscriber

- Install NuGet packages:
 - DotNetCore.CAP
 - DotNetCore.CAP.SqlServer
 - DotNetCore.CAP.RabbitMQ
- Add CAP services in Program.cs

```
builder.Services.AddDbContext<EventBusContextContext>(
    options => options.UseSqlServer(
        builder.Configuration.GetConnectionString("EventBusDBConnectionString")));
builder.Services.AddCap(options => {
    options.UseRabbitMQ("localhost");
    options.UseEntityFramework<EventBusContextContext>();
});
```

Publishing Events

Create an API endpoint that publishes an event using CAP

```
app.MapPost("/order", async (ICapPublisher capPublisher, Order order) => {
    await capPublisher.PublishAsync("OrderPlaced", order);
});
```

Subscribing to Events

Create Subscriber logic

```
public class OrderPlacedEventSubscriber: ICapSubscribe
{
    [CapSubscribe("OrderPlaced")]
    public async Task<IResult> Consumer(Order order)
    {
        //Handle the order placed event
    }
}
```

Add the EventSubscriber class as a service

```
builder.Services.AddScoped<OrderPlacedEventSubscriber>();
```

Configure RabbitMQ in Docker

- Install RabbitMQ as Docker Image and run as a container:

```
docker run -d --hostname my-rabbit --name ecomm-rabbit  
-p 15672:15672 -p 5672:5672 rabbitmq:management
```



Running and Testing

- Launch app
- Inspect databases
- Monitor RabbitMQ to see events and processing



Hands on Lab

Controllers and Actions

Objective:

Your goal is to reengineer the Estate Agent Property and Booking services, so they don't directly rely on each other when a property and any related bookings are deleted.

In this lab we will be working with and adapting code that focuses on the deletion of properties and any related bookings. The starter code uses a closely coupled approach and we need to reengineer things and create a loosely coupled solution that implements the event bus pattern using a couple of external packages, RabbitMQ and CAP.

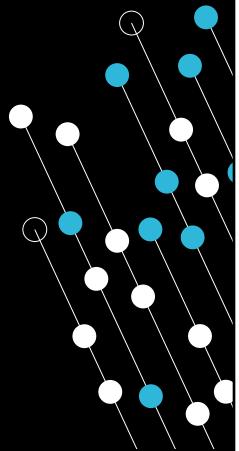
Summary

Know more about the what, where, why and when of minimal APIs

- Minimal APIs provide a streamlined approach to building APIs
- Ideal for small to medium-sized applications and microservices
- Faster development and improved performance

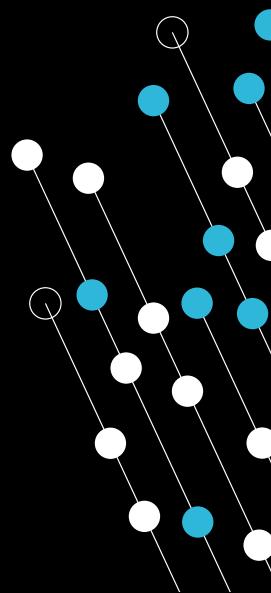
Advanced C#

An introduction to microservices and how to develop them using best practice



Advanced C#

Kubernetes



Session Objectives

To gain an understanding of howto use Kubernetes to automate the deployment, scaling, and management of containerized applications.



Session Content

- What is Kubernetes
- Kubectl
- Pods
- Nodes
- YAML Manifests
- Ports
- Environment Variables
- Pod Lifecycle
- Container Status
- Kubectl - Pods
- Services: Exposing Pods
- Service Types
- Controllers
- ReplicaSets
- Deployments
- Labels
- Namespaces
- Volumes
- ConfigureMaps



What is Kubernetes?

- Orchestration
- From <https://kubernetes.io...>
"Open-source software for automating deployment, scaling, and management of containerized applications"
- Originally built by Google
- Open and extensible
- From Greek κυβερνητης, pilot or Helmsman

Kubernetes (k8s) is an open-source container orchestration system for automating deployments of containerised applications.

Kubernetes has many solutions for managing and scaling very large quantities of containers by grouping them into manageable units. Because Kubernetes is open-source, it can be deployed as an on-premises, hybrid or public cloud solution.

Cloud services – such as Amazon Web Services, Microsoft Azure and Google Cloud Platform – offer managed Kubernetes services, which completely configure and deploy a Kubernetes cluster for you.

Kubernetes – A manager for your apps

Kubernetes helps you by:

- **Deciding where to run them**
- **Keeping them running**
- **Scaling**
- **Networking**
- **Updating Safely**



Kubernetes helps you by:

Deciding Where to Run Them: You tell Kubernetes what your app needs (like memory, CPU, or storage), and it finds the best place to run it.

Keeping Them Running: If something breaks, like a computer goes down, Kubernetes will restart your app somewhere else automatically.

Scaling: If more people use your app, Kubernetes can create more copies of it to handle the load. When things quiet down, it reduces the number of copies to save resources.

Networking: It makes sure your apps can talk to each other and to the outside world without you worrying about complex setup.

Updating Safely: If you want to update your app, Kubernetes does it one step at a time to avoid downtime.

In short: Kubernetes is your app babysitter. It makes sure your apps run where and how you need them, stay up, and adapt to changes—all automatically.

kubectl

To control and give instructions to a Kubernetes cluster, we need to interact with the cluster's API server.

The Kubernetes API can be easily interacted with using the kubectl CLI tool.

The command line interface for Kubernetes

```
kubectl [command] [TYPE] [NAME] [flags]
```

There are kubectl commands to:

- Run an image as a Pod in a cluster
- Create a Kubernetes object (e.g. a deployment)
- Get a list of running objects (e.g. Pods, Services, Deployments)
- Expose a Pod on a network (internal or external)
- Set properties of a resource
- Edit resources on a cluster
- Delete resources



kubectl

kubectl is the command-line interface for Kubernetes. Correctly configured, it enables a user with the correct permissions to make any API call against the Kubernetes cluster. There are other ways of communicating with Kubernetes, such as SDKs for most popular programming languages, a number of UIs with varying levels of functionality and cost, and we could of course hand-craft our API calls and make HTTPS requests. For this course, we'll stick with kubectl.

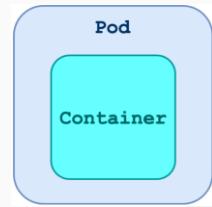
The general syntax for kubectl commands is:

```
kubectl [command] [TYPE] [NAME] [flags]
```

Where [command] is the action to be performed, [TYPE] is the type of resource, [NAME] is the name of the resource, and [flags] are additional options and parameters.

Pods

- The most basic unit of work in Kubernetes
- A single Pod consists of one or more containers.
- A pod should do one function, and it should do it well.
- The most common model is to have one container per Pod.



```
kubectl run the-pod --image=qa/myserverimage  
# create a pod, 'the-pod' running a 'home grown' image
```

A pod is the most basic unit of work in Kubernetes. Another way of thinking about what Kubernetes is to say it's a tool for managing pods. It is the basic building block of a Kubernetes application and can contain one or more containers. It abstracts away the container runtime and details what the workload looks like: which image(s) to use; which volume(s) (if any) to mount and where (more on volumes in a later module); which ports to expose; and any other additional configuration that the workload requires. Each pod gets its own unique IP address within the cluster, enabling communication between pods.

The most common model is to have one container per Pod. This is generally the desired configuration for a Pod, as one Pod will then perform a single process.

The simplest but least flexible way to create a pod is to use kubectl run:

Nodes and Control Planes - 1

- Pods in a Kubernetes cluster are hosted on **Nodes**.
- **Nodes** are the physical servers or VMs that comprise a Kubernetes Cluster.
 - There may be any number of **Nodes** in a cluster
- Every **node** has an agent known as a kubelet, which is a service that runs on the node to make sure that your applications and workloads are running and healthy.
- A **Control Plane** is responsible for communicating with and controlling **Nodes** and other Kubernetes objects in a cluster.

357

The Control Plane consists of several processes designed to manage different parts of the cluster, but, for the purposes of this course, the most important process to understand is the API server.

The API server communicates with each Node in the cluster via its kubelet agent (and vice versa).

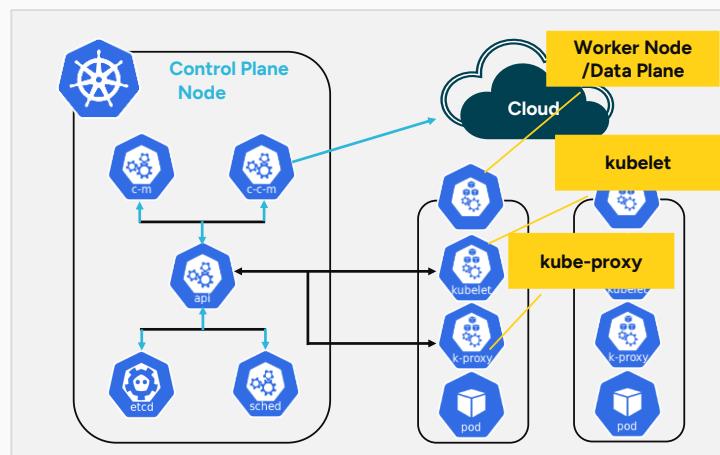
To control the state of the cluster, one has to call the API server. This is typically done using the CLI tool kubectl.

A cluster will typically have a control plane, that is used to communicate with it and tell it what to do, and a data plane where the actual work is performed.

It is entirely possible to have a single-node cluster which is both the control plane and the data plane.

Kubernetes architecture: All nodes

- Control Plane
 - Manages work
- Worker Nodes



Each node in the cluster including the control plane node, run the following three components:

- **kubelet:** The Kubernetes agent, receives instructions from the apiserver. Responsible for communicating with the container runtime to ensure that containers are running in a Pod.
- **kube-apiserver:** The API server is the control plane's front end, receiving and responding to API requests from users, the CLI (kubectl), and other components.
- **etcd:** A consistent and highly available key-value store that stores configuration data, state, and metadata for the entire cluster. The etcd data store is the 'source of truth' for the cluster's desired state.
- **kube-controller-manager:** Runs controller processes that regulate the state of various aspects of the system. Each kind in Kubernetes has a corresponding controller.
- **kube-scheduler:** Assigns pods to nodes based on resource requirements, affinity/anti-affinity rules, and other constraints. It is responsible for distributing work across the cluster.
- **cloud-controller-manager:** This component, if correctly configured will make API calls to the cloud environment if our cluster is running in the cloud. The

LoadBalancer service type we met briefly earlier involves calling out to the underlying cloud to create the cloud load balancer.

- **kube-proxy:** Maintains network rules on nodes, allowing communication between pods and external entities.
- **Container Runtime:** The software responsible for running containers, such as containerd, CRI-O, or less often nowadays, Docker.

YAML Manifests

- Configurations that are used to define the desired state of an object in a Kubernetes cluster.
 - Commonly written in YAML (could be JSON).
 - Easier to create multiple objects,
 - Can be maintained using version control.

```

kind: Deployment # what type of object are we defining
apiVersion: v1 # which API version the "kind" belongs to
metadata:
  name: buyerservice # a pod must have a name
spec: # the pod's specification
  containers: # a list of containers (just one in this case)
    - name: buyerservice # each container within the pod needs a name
      image: <docker-reg>/buyerservice:latest # what container image are we running
  
```

kubectl create -f the-pod.yaml

359

To define and create objects in Kubernetes, we typically use a YAML or JSON configuration file that specifies the object's details. In the case of a Pod, these details include container image(s), port(s), volumes, and other settings. kubectl can be used to create, inspect, and manage pods within a Kubernetes cluster.

This is the simplest-possible example of a pod definition in YAML:

```

kind: Deployment # what type of object are we defining (Deployments are
specialised pods combined with replicases)
apiVersion: v1 # which API version the "kind" belongs to
metadata:
  name: buyerservice # a pod/deployment must have a name
spec: # the pod's specification
  containers: # a list of containers (a list of one in this case)
    - name: buyerservice # each container within the pod needs a name
      image: qa/buyerservice:latest # what container image are we running.
      # (Name of image prefixed with docker hub user name)
  
```

kubectl create -f the-pod.yaml

This example creates a pod named "the-pod" with a single container ("the-container") using the specified container image. By default, Kubernetes will attempt to pull images from Docker Hub, unless a container registry has been

provided as part of the image name.

YAML Manifest Mappings

Section	Explanation
apiVersion	The version of the API, e.g. v1. This will differ between objects and it is best to refer to the documentation if you are unsure which to use.
kind	The type of object we are defining, such as pod, deployment or service.
metadata	Provides information about the object, such as the name of the object and the labels attached to it.
spec	The specification for the object itself. For a Pod, this would include information about its containers, volumes, hostname, scheduling and more.

Create the objects defined in a manifest using **kubectl** with the **-f** argument:

```
# kubectl create -f [TEMPLATE_FILE]  
kubectl create -f nginx.yaml
```

For more see: <https://kubernetes.io/docs/home/>

360

The best place to go to find a manifest template for a Kubernetes object is the official documentation (<https://kubernetes.io/docs/home/>).

Ports

- Defining a port mapping in the container spec provides information about what port the containerised application is accessible on.
- Setting the port is not required to expose the Pod to the network
- this is achieved using a Service.

```
apiVersion: v1
kind: Deployment
metadata:
  name: mymicroservice
  labels:
    app: mymicroservice
spec:
  containers:
    - name: mymicroservice
      image: <docker-reg>: mymicroservice
      ports:
        - containerPort: 80
```

Environment Variables

- Many applications require environment variables for configuration at runtime.
- This can be used in Pod templates in Kubernetes by using the env property:

```
apiVersion: v1
kind: Pod
metadata:
  name: mymicroservice
  labels:
    app: mymicroservice
spec:
  containers:
    - name: mymicroservice
      image: qa/mymicroservice
      env:
        - name: MY_VAR
          value: "value of my var"
```

362

Environment variables may include entries like the following:

env:

```
- name: ASPNETCORE_ENVIRONMENT
  value: "Production"
- name: ASPNETCORE_URLS
  value: http://*:3011
- name: ConnectionStrings_sqlestateagentdata
  value:
"Server=sqlestateagentbuyerdata;Database=EBuyer;
User
Id=sa;Password=PaSSwOrdPaSSwOrd;MultipleActiveR
esultSets=true;Encrypt=False;TrustServerCertificate=T
rue"
```

Pod Lifecycle

- The Status of a Pod is a PodStatus Object in the Kubernetes API.
- It provides information about the current status and health of the Pod.

Value	Description
Pending	Kubernetes has accepted the request and has started downloading the required images and is in the process of starting the containers.
Running	The Pod is bound to a node, all the Containers have been created and are in the process of starting or restarting.
Succeeded	All containers have executed without error, have now exited and will not be restarted.
Failed	At least one of the containers have terminated with a non zero status.
Unknown	For some reason the phase is unobtainable, if the Node which the Pod is running on can't be accessed for example.



Container Status

The health of the Pod depends on the health of the containers within the Pod.

Containers can either be

- Waiting

```
State: Waiting  
Reason: ImagePullBackoff
```

- Running

```
State: Running  
Started: Tue, 04 Feb 2020 14:55:08 +0000
```

- Terminated

```
State: Waiting  
Reason: CrashLoopBackoff  
Last State: Terminated  
Reason: Error  
Exit Code: 1  
Started: Tue, 04 Feb 2020 15:13:07 +0000  
Finished: Tue, 04 Feb 2020 15:13:07 +0000
```

364

If an image is not accessible, then there may be an error (ImagePullBackOff) displayed as the Reason whilst still in a waiting state.

When a container is running there will be some information to show when the container was started.

A Terminated state indicates that the container has stopped, the exit status will be displayed.

Kubectl - Pods

- **View the Pods in a cluster:**

```
kubectl get pods
```

- **Get further info about Pods:**

```
kubectl describe pods
```

```
kubectl describe pod mypod
```

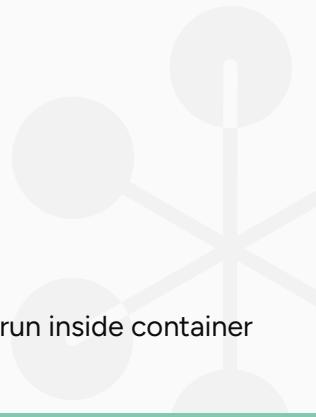
- **Delete a Pod by its name:**

```
kubectl delete pod mypod
```

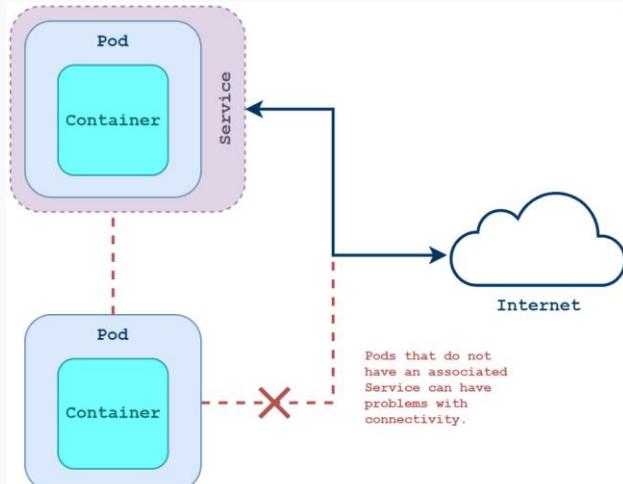
- **Gain access to a container:**

```
Kubectl exec mypod -- echo hi
```

-- Used to separate kubectl command from command to run inside container



Services -Exposing Pods



By default, when a Pod is created, it is connected to the pod network and given a private IP address.

This can cause issues when trying to get that Pod to communicate with other things, since Pods are ephemeral (they are often deleted and made again) each time they are created, they are given a new private IP address.

A better way to communicate with a Pod is to attach it to a **Service**.

Services are used to expose Pods to other networks.

This could be within the cluster, such as using a *ClusterIP* service, or to the internet, using a *NodePort* or *LoadBalancer* service.

Services also allow the ephemeral Pods to always be accessed from the same address.

Service Types

Provide a consistent way to expose and access applications within a cluster:

- ClusterIP
- NodePort
- LoadBalancer

```
kubectl expose deployment the-app --port=80
# or:
kubectl expose deployment the-app --port=80 --type=NodePort
# or:
kubectl expose deployment the-app --port=80 --type=LoadBalancer
```

To expose a Pod to a network, it needs to be associated with a Service object.

Different types of Services define different networking rules and capabilities. For example, a ClusterIP-type Service will only expose a Pod to other Pods within the cluster, whereas a LoadBalancer-type Service will expose a Pod to an external network.

Pods within a cluster are assigned internal IP addresses. A Kubernetes Service is an abstraction that defines a logical set of pods and a means by which to access them. Services provide a consistent way to expose and access applications within a cluster, enabling communication between different parts of an application or between different applications.

Services allow the use of hostname resolution to resolve names to IP addresses, allowing applications to refer to the name of the Service as a network location in order to communicate with each other. As pods come and go, as they're being deleted and recreated by ReplicaSets, their individual IP addresses will change. The service endpoint and name will remain the same. This is what enables other workloads in the cluster to interact with the service without needing to know the individual pod IP addresses.

Basic service types:

- ClusterIP: The default type. It exposes the service on a cluster-internal IP and makes it accessible only within the cluster. It's a sensible default as most applications have only one front-end, but multiple backend services.
- NodePort: Exposes the service on each node at a high-numbered static port. The service is accessible externally using the node's IP address and the assigned port. Creating a NodePort also creates a ClusterIP for the service, so it's still available internally via its name.

An Example Service Manifest

```
apiVersion: v1
kind: Service
metadata:
  name: mymicroservice
spec:
  selector:
    app: mymicroservice
  ports:
    - protocol: TCP
      port: 80
```



Let's break this manifest down further.

```
apiVersion: v1
kind: Service
```

This section defines that we are creating a Service object according to the

```
apiVersion: v1 specification.
```

```
metadata:
```

```
  name: mymicroservice
```

Defining the name: mapping under the metadata: section gives the Service a name. Pods within the cluster will then be able to refer to this network location using this name.

```
spec:
```

```
  selector:
```

```
    app: mymicroservice
```

```
  ports:
```

```
    - protocol: TCP
```

```
      port: 80
```

The spec: section provides the Service's configuration.

The selector: mapping is used to associate the Service with another object, such as a Pod. In the above example, the Service will direct any network traffic sent to the Service to any Pod with the label app: mymicroservice.

The ports: mapping specifies which ports the Service will expose. The above example will direct any traffic to the Service on port 80 to the Pod on port 80.

If you wanted the exposed port and the target port (the Pod's port) to be different, you could also specify the targetPort:

```
ports:
```

```
  - protocol: TCP
```

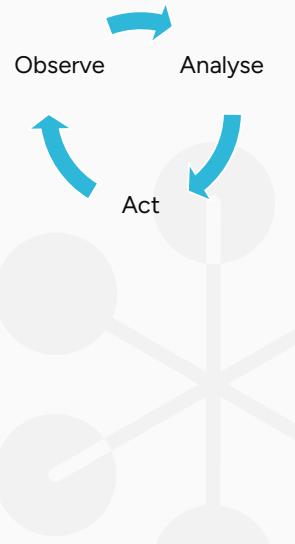
```
    port: 80
```

Controllers / Operators / Watch-loops

Work to bring the actual state of the cluster closer to the desired state

Examples of standard Kubernetes controllers:

- ReplicaSet
- Deployment
- Node
- Service
- Namespace



The three terms are used interchangeably and ultimately mean the same thing.

Controllers are control loop processes that continuously observe the state of the cluster and work to bring the actual state closer to the desired state. Controllers operate on a control loop, where they constantly reconcile the current state of resources with the desired state specified by users. This ensures that the system remains in the desired state and responds to changes over time.

Examples of standard Kubernetes controllers include:

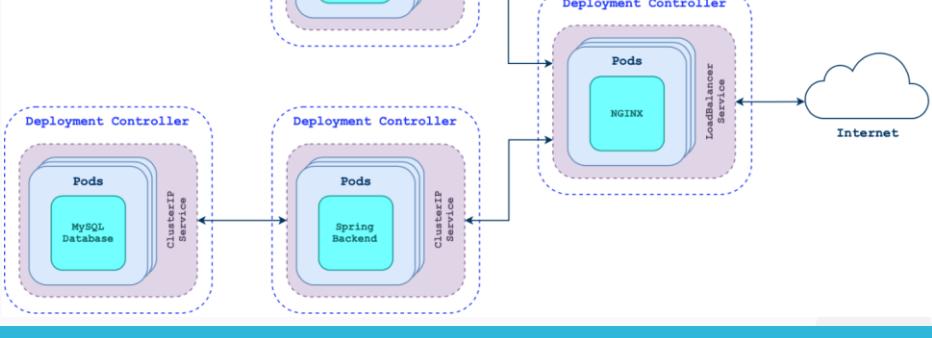
- **ReplicaSet:** Ensures that a desired number of pods are running at all times.
- **Deployment:** Manages ReplicaSets and enables rolling updates and rollbacks of application versions.
- **Node:** Ensures the correct number of nodes are available in the cluster. It monitors node status and takes action if a node becomes unreachable or is marked for termination.
- **Service:** Ensures that the desired service endpoints are available.

- Namespace: Manages the creation, update, and deletion of namespaces.

Controllers

Deployment Controllers are used to create and manage replicas of Pods.

It will constantly monitor how many replicas are currently running and update the current state according to the desired state.



Kubernetes is designed to manage and deploy containers (as Pods) on a massive scale.

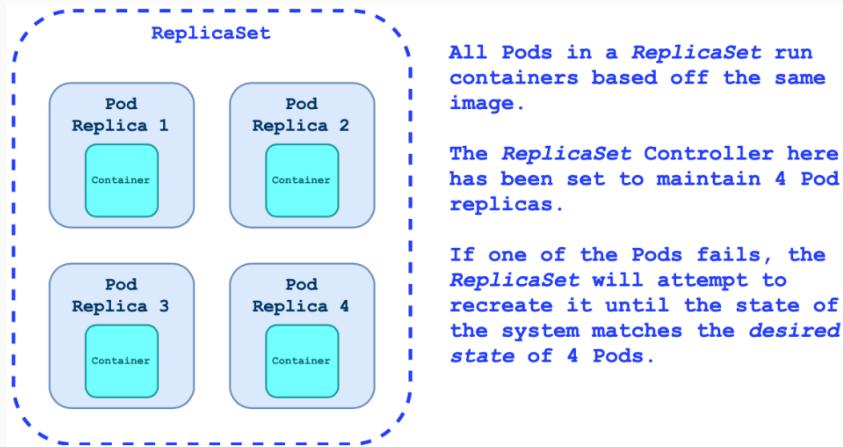
As a result, one rarely creates and manages the deployment of an application using individual Pods.

Controllers are used to manage Kubernetes objects at scale by defining a desired state for the system.

A Controller will constantly check the current state of the system and automatically work to pull it into the desired state.

This process of constant checking is known as a control loop, and allows our Kubernetes clusters to self-manage with minimal human oversight

ReplicaSets



374

ReplicaSets manage and maintain a set of identical Pods replicas.

Replicating Pods provides higher availability and resiliency to our containerised applications.

You can easily scale a ReplicaSet in and out with the command:

```
kubectl scale --replicas=<NUMBER_OF_REPLICAS> replicaset/<REPLICASET_NAME>
```

ReplicaSets are **not** suited for collections of Pods whose configurations are likely to change, such as for a rolling update. In this case, you should use a [Deployment](#).

Note: the [Kubernetes documentation](#) actually recommends using Deployments over ReplicaSets in most scenarios, as their configurations make use of ReplicaSets anyway while providing more functionality.

Deployments

- Deployments are very similar to ReplicaSets, and in fact are extensions of the ReplicaSet Controller.
- They manage and maintain replicas of Pods to provide availability and resiliency.
- The added benefit of a Deployment is its ability to manage rolling updates.
 - A rolling update is the process of gradually updating an application with a new version in order to eliminate service downtime.
 - During a rolling update, there is a period where both the old and the new version of the application will be running simultaneously.
 - This means that users can continue to use the application, even while it is in the process of being replaced with the new version.
- Can define specific deployment strategies and configure how they behave, such as specifying:
 - **maxSurge** – the maximum number of extra Pods that can be provisioned during the rolling update.
 - **maxUnavailable** – the maximum number of Pods that can be unavailable during the rolling update.

375

An example "Deployment" manifest:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: myshop-deployment
  labels:
    app: myshop
spec:
  replicas: 10
  selector:
    matchLabels:
      app: myshop
  strategy:
    rollingUpdate:
      maxSurge: 10%
      maxUnavailable: 25%
  template:
    metadata:
      labels:
        app: myshop
  spec:
    containers:
      - name: myshop
        image: <docker-reg>/myshop-api
        ports:
          - containerPort: 80
```

Labels – ReplicaSet / Deployment

Key/value pairs used by Kubernetes controllers to manage objects.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: the-app
spec:
  replicas: 3
  selector:           # how the RS can find the pods
    matchLabels:
      app: the-app
  template:
    metadata:
      labels:          # every pod will have this label
        app: the-app
    spec:
      containers:
        - name: the-container
          image: public.ecr.aws/docker/library/httpd
```

Labels are key-value pairs that are attached to objects (such as pods, services, or nodes) to provide metadata and enable the organization and selection of resources.

Labels are a fundamental component of Kubernetes for grouping and categorising resources.

Kubernetes controllers use label selectors to identify the resources they manage.

The selector is specified in the object's configuration and is used to match against the labels of the pods.

Here's that Deployment manifest from earlier again:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: the-app
spec:
```

```

replicas: 3
selector:          # how to find the pods
  matchLabels:
    app: the-app
template:
  metadata:
    labels:      # every pod will have this label
      app: the-app
spec:
  containers:
    - name: the-container
      image: public.ecr.aws/docker/library/httpd

```

As long as there are three pods with the label app with a value of the-app, the ReplicaSet controller will be happy. If at any time there are fewer than three, new pods will need to be created.

And here's a YAML manifest for a Service object:

```

apiVersion: v1
kind: Service
metadata:
  name: the-service
spec:
  selector:          # how to find the pods
    app: the-app
  ports:
    - protocol: TCP
      port: 80

```

In this example, the service the-service selects pods with the label app: the-app and exposes port 80. Any other pods within the cluster can access this service using its DNS name or IP address and port 80.

Labels can be used to attach metadata, such as application names, environment types, or ownership labels, to Kubernetes objects.

Labels can also be used by us to select resources.

Namespaces

A Namespace in Kubernetes is effectively a virtual cluster; it is a way to split up clusters into smaller sub-clusters.

Resources can then be assigned to a Namespace to organise them.

- Resources in a cluster must have unique names within their Namespace, meaning two resources can have the same name if they are in different Namespaces.
- Namespaces cannot be nested within other Namespaces.
- Resources can only exist in one Namespace.
- Namespaces are not fully isolated from each other, they are just hidden, this means that Services can still communicate across Namespaces if required.

Namespaces

- Logically-isolated subclusters

```
# create 3 namespaces
kubectl create namespace development
kubectl create ns test
kubectl create ns production
# create the same deployment in each namespace
kubectl create -f the-app.yaml --namespace=development
kubectl create -f the-app.yaml --namespace=test
kubectl create -f the-app.yaml -n production
# expose the deployment in each namespace
kubectl expose deployment the-app -n development
kubectl expose deployment the-app -n test
kubectl expose deployment the-app -n production

kubectl get pods --namespace=production
kubectl get pods -n test
kubectl get pods --all-namespaces
kubectl get pods -A
```

A Namespace is a logically-isolated subcluster, potentially with separate security settings and different resource quotas.

If we have three different environments, for example development, test, and production, we could have three separate clusters to get complete isolation. But, that's then three clusters we have to maintain and more importantly pay for. We could achieve much the same effect using three (or more) different namespaces. There would still be an administrative burden to maintain the security settings and quotas, but we'd have that anyway with multiple clusters.

However, many organisations prefer the clear separation of running multiple clusters.

The above example will create three ClusterIP services with three different addresses in the three different namespaces.

Volumes

- An abstraction of a file system
- Can be shared
- Attached to pods
- Mounted in containers
- Many types available



A Volume is an abstraction of a file system used by Kubernetes. Data in a volume can be shared among all the containers in a pod. Volumes provide a way for containers to persist and share data, even if they are running on different nodes in the cluster. Volumes are attached to pods and mounted in containers and the data within them is retained across container restarts and possibly for even longer. Volumes can be mounted as read-only or read-write (the default).

Volumes abstract away the underlying storage details, providing a uniform way to work with volumes. This allows us to switch between different storage solutions without modifying the application.

Volume types

Common types:

- emptyDir
- hostPath
- persistentVolumeClaim

Container Storage Interface (CSI) Volumes:

- Follow the Kubernetes principle of extensibility
- Administrators need to install the appropriate driver for their Vendor

Kubernetes supports various types of volumes, allowing us to choose the most appropriate storage solution for their applications. Some common types include:

- **emptyDir**: An empty directory that is created when a pod is assigned to a node and exists as long as that pod is running. EmptyDirs are deleted when the pod is deleted
- **hostPath**: A volume that mounts a file or directory from the host node's filesystem into the pod.
- **persistentVolumeClaim**: Useful for stateful workloads like databases. Beyond the scope of this course.

Mounting Volumes

- Add **volumes** to the pod spec
- Add **volumeMounts** to the container(s)
- Specify a **mountPath** within the container

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  volumes:          # a list of volumes
  - name: a-volume
    emptyDir: {}
  containers:
  - name: the-container
    image: httpd
    volumeMounts: # where to mount the volume(s)
    - name: a-volume
      mountPath: /data
```

383

Here's an example of using an EmptyDir volume in a pod definition:

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  volumes:          # a list of volumes
  - name: a-volume
    emptyDir: {}
  containers:
  - name: the-container
    image: httpd
    volumeMounts: # where to mount the volume(s)
    - name: a-volume
      mountPath: /data
```

In this example, an EmptyDir volume named 'a-volume' is mounted at the path

'/data' within the container 'the-container'. Any data written to this volume can be shared among all containers within the same pod if they have it mounted.

Debugging Kubernetes Clusters

Bridge to Kubernetes

- Allows you to work with Kubernetes clusters directly from within Visual Studio (and VSC)
- Can run and debug code on local machine while using the Kubernetes services running in remote clusters
- Gives you the advantages of working locally as well as the ability to integrate and test changes in the context of a full Kubernetes environment
- **Unfortunately**, it does NOT support Docker for Desktop Kubernetes clusters and is designed to work primarily with managed Kubernetes services such as Azure Kubernetes Services (AKS).
- For more see <https://learn.microsoft.com/en-us/visualstudio/bridge/bridge-to-kubernetes-vs>

386

"Bridge to Kubernetes" is a development tool provided by Microsoft that enables developers to work with a Kubernetes cluster directly from their local development environment. It allows developers to run and debug code on their local machine while seamlessly connecting to services running in a remote Kubernetes cluster. This provides the advantages of local development (quick iteration, access to local tools) combined with the ability to integrate and test changes in the context of a full Kubernetes environment. Unfortunately, it does **not** support Docker for Desktop Kubernetes clusters.



Hands on Lab

Controllers and Actions

Objective:

Your goal is to use Kubernetes to configure the Estate Agent Microservices ready for deployment

In this lab you will configure the image names in docker-compose.yml file to prepare them for deployment to Docker Hub. Then you will create a set of YAML files, one for each service, with configuration settings that will be used in a Kubernetes deployment. You will then invoke the relevant kubectl commands to carry out the deployment. Finally you will test the endpoints to ensure everything works as expected.

Summary

Now Know:

- What is Kubernetes
- Kubectl
- Pods
- Nodes
- YAML Manifests
- Ports
- Environment Variables
- Pod Lifecycle
- Container Status
- Kubectl - Pods
- Services: Exposing Pods
- Service Types
- Controllers
- ReplicaSets
- Deployments
- Labels
- Namespaces
- Volumes
- ConfigureMaps



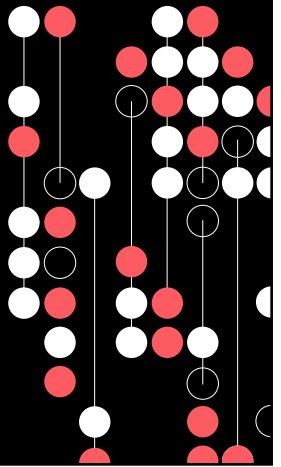


Advanced C#

An introduction to microservices and how to develop them using best practice

Advanced C#

Microservice Security



Session Objectives

Understand how to ensure ASP.NET code and microservices are operating in a secure manner

Session Content

- ASP.NET API Security using Jason Web Tokens
- Docker Security
- Kubernetes Security

ASP.NET API Security

- Authorisation
- JSON Web Tokens (JWT)



Classic Authorisation

- When user logs in, a cookie is returned from the server and stored in their web browser
- That cookie is checked on subsequent requests
- Authorisation can be applied to whole controllers, or individual actions
- Simple Authorisation
 - Users simply have to be authenticated to access a controller or action
- Claims-Based Authorisation
 - Users must have a specific claim to access a controller or action
 - Can specify values that those claims must have

```
[Authorize]
3 references
public class HomeController : Controller
{
```

User must have "Admin" claim, with values 2, 3 or 4

```
builder.Services.AddAuthorization(options =>
{
    options.AddPolicy("SuperAdmin",
        policy => policy.RequireClaim("Admin", "2", "3", "4"));
});
```

```
[Authorize("SuperAdmin")]
3 references
public class HomeController : Controller
{
```

The JWT Authorisation Scheme

- ASP.NET uses Microsoft Identity, together with cookies, as its default authorisation schema
- But you can use a different schema if you want
- A common example is JWT – JSON Web Tokens
 - An open, industry standard method of handling authentication
 - Server sends the client a token
 - Client sends the token back with each request
 - Token is encrypted. If the server can decrypt it, we know it was encrypted with the correct key
 - Tokens include claims, which are added before encryption
 - Frequently used with Web API

Enabling the JWT Authorisation Scheme

- To change from the default authorisation scheme to JWT:

```
builder.Services.AddAuthentication()
    .AddJwtBearer(options =>
{
    options.RequireHttpsMetadata = false;
    options.SaveToken = true;
    options.TokenValidationParameters = new TokenValidationParameters
    {
        ValidIssuer = "http://myorg.com",
        ValidAudience = "http://myorg.com",
        IssuerSigningKey = new SymmetricSecurityKey
            (Encoding.UTF8.GetBytes("MyApplicationsecret"))
    };
});
```

- Will authorise requests if they are sent with a valid JWT token, signed by <http://myorg.com>, with the correct key

- To generate those tokens:

```
var token = new JwtSecurityToken();
string tokenText = new JwtSecurityTokenHandler().WriteToken(token);
```

Note that this slide shows the issue and key being hard-coded into `ConfigureServices()`. This is not appropriate for real-world scenarios. These details (especially the key) should be considered to be an application secret and should be stored using an appropriate level of security.

Docker Security

- The inherent security of the kernel
- The attack surface of the Docker daemon
- Vulnerabilities in container configuration profiles
- The impact of kernel “hardening” security features



Docker Security

- When assessing Docker security, it is essential to focus on the following key areas:
- The inherent security of the kernel, including its support for namespaces and cgroups.
- The attack surface of the Docker daemon.
- Potential vulnerabilities in container configuration profiles, whether they are default or customized by users.
- The impact of kernel “hardening” security features on container interactions.

When you start a container with docker run, behind the scenes Docker creates a set of namespaces and control groups for the container.

Namespaces provide the first and most straightforward form of isolation. Processes running within a container cannot see, and even less affect, processes running in another container, or in the host system

Control Groups are another key component of Linux containers. They implement resource accounting and limiting. They provide many useful metrics, but they also help ensure that each container gets its fair share of memory, CPU, disk I/O; and, more importantly, that a single container cannot bring the system down by exhausting one of those resources.

The inherent security of the kernel

- **Namespace Isolation:** Ensuring proper isolation between containers using namespaces (such as PID, network, and mount namespaces) is crucial. Misconfigurations can lead to unintended interactions and security breaches.
- **Cgroup Management:** Control groups (cgroups) allow resource allocation and limits for containers. Mismanagement can result in resource exhaustion (denial of service attacks) or privilege escalation.
- **Kernel Vulnerabilities:** The kernel itself must be secure. Unpatched vulnerabilities can be exploited by attackers to compromise containers.

The attack surface of the Docker daemon

- **Rootless Mode:** Enable rootless mode where possible. By default, the daemon requires root privileges but running in rootless mode restricts the daemon's privileges and enhances security.
- **Regular Updates:** Keep Docker updated to patch vulnerabilities and improve security.
- **Exposing the Daemon Socket:** Avoid exposing the Docker daemon socket directly to the internet. If necessary, use TLS (HTTPS) to protect it

400

Running containers (and applications) with Docker implies running the Docker daemon. This daemon requires root privileges unless you opt-in to [Rootless mode](#)

Kubernetes Security

- Three “realms”
- Role-Based Access Control (RBAC)
- Network Policy
- Pod Security



The three “realms” of security

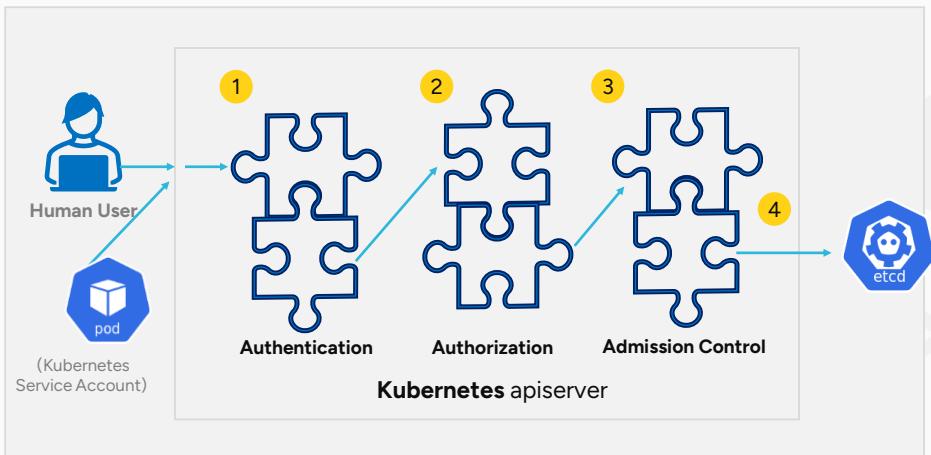
- Role-Based Access Control
- Pod Security Policy
- Network Policy



There are three ‘realms’ of security in Kubernetes:

- **Role-Based Access Control (RBAC):** Defines which actors are allowed to perform which actions (in which namespaces).
- **Pod Security Policy:** Controls how much access our pods (and their containers) have to our hosts.
- **Network Policy:** Defines what can communicate with our pods and how.

Accessing the API



To perform any action in a Kubernetes cluster, an entity goes through three main steps:

- **Authentication:** Credentials are checked. Is this entity known to the system?
- **Authorisation:** RBAC. Does this entity have permission to perform this action?
- **Admission Control:** Where policy can be applied? Yes, this entity can perform this action, but only under certain circumstances.

These steps are described in more detail in the official documentation at <https://kubernetes.io/docs/admin/accessing-the-api/>

Role Based Access Control (RBAC)

- Roles and ClusterRoles
- Rules
- Resources and Operations (verbs)
- RoleBindings and ClusterRoleBindings
- Subjects



Role-Based Access Control (RBAC) enables fine-grained control over who can perform which actions (such as creating, updating, or deleting resources) within a cluster. RBAC allows administrators to define roles and role bindings, specifying what permissions users, groups of users, or service accounts have in the cluster. This helps enforce the principle of least privilege, ensuring that users only have the permissions necessary for their specific tasks.

Roles

A Role is a set of rules that define a set of permissions (rules) for performing operations on specific resources in a particular namespace. Roles are scoped to a namespace and are used to grant access to resources within that namespace only.

ClusterRoles

A ClusterRole is just like a Role but operates at the cluster level, providing permissions across all namespaces.

RoleBindings

A RoleBinding binds a Role or a ClusterRole to a user, group, or service account

within a specific namespace. It specifies which users or entities have the permissions defined by the associated Role.

ClusterRoleBindings:

A ClusterRoleBinding operates at the cluster level. It binds a ClusterRole to a user, group, or service account across all namespaces.

Subjects

A Subject is an entity that can be assigned permissions. Subjects are referenced in RoleBindings and ClusterRoleBindings. Subjects are either:

- a User
- a Group or
- a ServiceAccount

ServiceAccounts

A ServiceAccount is a Kubernetes object that provides an identity for processes running in a Pod. If we have a Pod-based workload that needs to be able to communicate with the apiserver, to take actions within the cluster, we need to use a service account. By default, service accounts have no permissions.

RBAC Workflow

- Determine which namespace or create if necessary
- Define Role(s) or ClusterRole(s)
- Create RoleBinding(s) or ClusterRoleBinding(s)
- Verify permissions



- Determine which namespace (if any) the permissions need to apply to or possibly even create the namespace.
- Define Roles and / or ClusterRoles with the correct permissions.
- Create RoleBindings and / or ClusterRoleBindings.
- Verify permissions.

Users can only perform actions for which they have been granted explicit permissions.

RBAC Components

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: pod-reader
rules:
- apiGroups: [""]
  resources: ["pods"]
  verbs: ["get", "list"]
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: RoleBinding
metadata:
  name: pod-reader-binding
  namespace: some-namespace
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

```
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: pod-reader-binding
  # no namespace is specified
subjects:
- kind: User
  name: alice
  apiGroup: rbac.authorization.k8s.io
roleRef:
  kind: Role
  name: pod-reader
  apiGroup: rbac.authorization.k8s.io
```

In this example:

The ClusterRole named ‘pod-reader’ allows subjects to perform ‘get’ and ‘list’ operations on pods anywhere in the cluster. The RoleBinding named ‘pod-reader-binding’ binds the ‘pod-reader’ ClusterRole to the user ‘alice’ in the some-namespace namespace only.

Finally, the ClusterRoleBinding named ‘pod-reader-cluster-binding’ binds the ‘pod-reader’ Role to the user ‘alice’ across the entire cluster.

NOTE: If the intention is to give ‘alice’ pod-reader permissions across the entire cluster, then the RoleBinding is superfluous. This is intended to be an example only.

Verifying permissions

```
kubectl auth can-i get pods --as alice  
yes  
kubectl auth can-i create pods --as alice  
no
```

It might be desirable to check the RBAC settings. We can use kubectl to do this:

```
kubectl auth can-i get pods --as alice  
yes  
kubectl auth can-i create pods --as alice  
no
```

Network policies

- All traffic is allowed by default
- Network plug-in needs to support Network Policies (not all do)
- Ingress (inbound) and Egress (outbound) rules
- Identify source / destination by Pod labels, Namespace labels or IP address ranges

NetworkPolicies ('netpol') allow us to define rules that control the traffic between pods. We use them to specify which pods are allowed to communicate with each other within the cluster, using which protocols and which port numbers. This adds another a layer of network security to our existing controls (network firewalls / security groups, network access control lists (NACLs), firewall appliances, and host-based firewalls).

In order to use netpols, the cluster networking plugin must support them. Many plugins do but some, notably the AWS VPC CNI plugin, do not.

NetPols support both ingress and egress rules. Ingress policies control inbound traffic to a set of pods, while Egress policies control outbound traffic from a set of pods.

Like all objects in Kubernetes, network policies use label selectors to specify which pods the policy applies to. These can be Pod labels (PodSelector) or Namespace labels (NamespaceSelector). Netpols can also use IP address range-based rules (ipBlock).

Network policies consist of a set of rules which each define the allowed traffic based on specified criteria such as pod labels, namespaces, IP blocks, and ports. Rules are processed in order, and the first rule that matches is applied.

Isolated and Non-Isolated Pods

- **By default, pods are non-isolated**
 - i.e. They accept traffic from any source.
- **Pods become isolated by having a NetworkPolicy that selects them.**
 - Once there is a NetworkPolicy in a namespace selecting a particular pod, that pod will reject any connections that are not allowed.
- **It is worth noting that network policies don't conflict.**
 - If any policy selects a pod, the pod is restricted to what is allowed by the union of those policies' ingress/egress rule

An example NetworkPolicy may look like the following:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
  ports:
  - protocol: TCP
    port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
  ports:
  - protocol: TCP
    port: 5978
```

Default deny

- If no network policies are defined, Kubernetes follows a '**default allow**' approach
- There is **no** 'deny' action
- A policy that applies to all pods and defines no rules, effectively says no inbound traffic is permitted

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {} # applies to all pods
  policyTypes:
  - Ingress
  # no rules so no traffic matches
  # so no traffic permitted
  # No "Egress" type so all outbound allowed
```

If no network policies are defined, Kubernetes follows a 'default allow' approach, meaning that all traffic between all pods is allowed. Network policies allow us to explicitly define the allowed communication paths. If using netpols, it might be considered a good practice to create a blanket 'default-deny' policy on cluster creation and then to explicitly allowlist permitted traffic. Whilst we can't explicitly deny traffic in a netpol (there is no 'deny' action), we can create a policy that applies to all pods and defines no rules, effectively saying no inbound traffic is permitted.

Here is an example of a default-deny policy:

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny-all
spec:
  podSelector: {} # applies to all pods
  policyTypes:
  - Ingress
  # no rules so no traffic matches
```

```
# so no traffic permitted  
# No "Egress" type so all outbound allowed
```

Pod Security

- Control privileges at the container, pod, namespace, or cluster level
- Very few workloads need to run as root
- Many do!



A lot of the publicly-available base images run as the root user, giving them far more permissions than they actually need. We can use security contexts to limit the permissions that a pod or its containers has on the host. We can use the Pod Security Admission controller to control this at a namespace level and, with some advanced configuration, at the cluster level.

Pod Security Context

Control privileges at the pod or container level.

Common Security Context Fields:

- runAsNonRoot
- runAsUser
- runAsGroup
- fsGroup
- capabilities

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  containers:
    - name: the-container
      image: the-image
```

A Pod Security Context allows us to define how the containers within the pod run and interact with the underlying system. These settings include options related to user privileges, Linux capabilities, access to host namespaces, and more.

The Pod Security Context can be applied at the pod level - meaning that the settings specified in the security context apply to all containers within the pod - or at the container level, allowing for per-container customisation.

Common Security Context Fields:

- **RunAsNonRoot:** A simple true / false setting that enforces that this workload must not run with root privileges.
- **RunAsUser:** A specific UID (User ID) to run as. It can be a numeric UID or a username.
- **RunAsGroup:** A specific GID (Group ID) to run as. It can be a numeric GID or a group name.
- **FSGroup:** Specifies the GID for volumes attached to the pod or mounted in containers.
- **Capabilities:** Defines Linux capabilities for the containers, which provide fine-grained control over the privileges of a process. We can add or drop specific capabilities as needed.

There are various other settings possible within a security context, relating to Security Enhanced Linux (SELinux), enforcing a read-only file system, denying privilege escalation, Secure Computing (Seccomp) profiles, and Sysctls. These are advanced topics.

```
apiVersion: v1
kind: Pod
metadata:
  name: the-pod
spec:
  securityContext:
    runAsUser: 1000
    fsGroup: 2000
  containers:
    - name: the-container
      image: the-image
```

The `runAsUser` field specifies that the container must run with the UID 1000. The `fsGroup` field specifies that the mounted volumes in the pod belong to the group with an ID of 2000.

Security contexts are a crucial aspect of container security, helping to ensure that containers operate with the appropriate level of isolation and least privilege.

It's important to carefully configure security contexts based on the specific security requirements of your application.

Pod Security Admission

controller allows cluster administrators to enforce security policies during pod creation

Applied to the namespace via labels

Three policies:

- privileged – no restrictions
- baseline – minimal restrictions
- restricted – follow current hardening best practices

Two levels of enforcement:

- enforce
- warn

```
apiVersion: v1
kind: Namespace
metadata:
  name: restricted-namespace
labels:
  pod-security.kubernetes.io/enforce: baseline
  pod-security.kubernetes.io/enforce-version: latest
  pod-security.kubernetes.io/warn: restricted
  pod-security.kubernetes.io/warn-version: latest
```

421

This is an example of an Admission Controller that enforces policy. The Pod Security admission controller allows cluster administrators to enforce security policies during pod creation. It ensures that pods adhere to the specified security constraints.

There are three types of policy:

- Privileged – no restrictions.
- Baseline – minimal restrictions.
- Restricted – follow current hardening best practices.

And two levels of enforcement:

- Enforce - do not allow any pods to be created which violate the policy .
- Warn - provide a warning if a pod will violate the policy, but allow it be created anyway.

Policies are applied to a namespace via labels and to the cluster by passing a special configuration file to the kubelet by a cluster administrator.

Here's an example of a namespace-level policy:

```
apiVersion: v1
kind: Namespace
metadata:
  name: my-restricted-namespace
labels:
  pod-security.kubernetes.io/enforce: baseline
  pod-security.kubernetes.io/enforce-version: latest
  pod-security.kubernetes.io/warn: restricted
  pod-security.kubernetes.io/warn-version: latest
```

In this example, it is possible to create pods in the ‘restricted-namespace’ that violates the ‘restricted’ policy, with warnings, but not possible to create pods that violate the ‘baseline’ policy. For the latest versions of these policies, see:
<https://kubernetes.io/docs/concepts/security/pod-security-standards/>

Hands on Lab

Controllers and Actions

Objective:

Your goal is to configure the Estate Agent application to operate in a secure manner by requiring clients to authenticate and provide a JSON Web Token when requesting application services.

You will alter the logic of the Estate Agent Microservices SellerService project so that none of its methods can be called unless the user's credentials have been authenticated. Proof of successful authentication will be demonstrated in the passing and receipt of a Jason Web Token (JWT).

Summary

- All the usual security measures (authentication, authorisation, https, etc.) still apply for frontend web sites.
- JWT are a good way of authenticating and authorising requests made to an API microservice.
- Docker containers are inherently secure, particularly when processes run as non-privileged users within the container and you keep up to date with security patches..
- Use RBAC to restrict access to the Kubernetes API
- Store sensitive information in Kubernetes secrets and encrypt them.
- Define Network Policies to control traffic between Kubernetes pods
- Apply Pod Security Standards (e.g., security contexts, Pod Security Policies)
- Regularly update Kubernetes and its components to patch vulnerabilities.