.NET Core

.NET Core/ASP.NET Core/EF Core

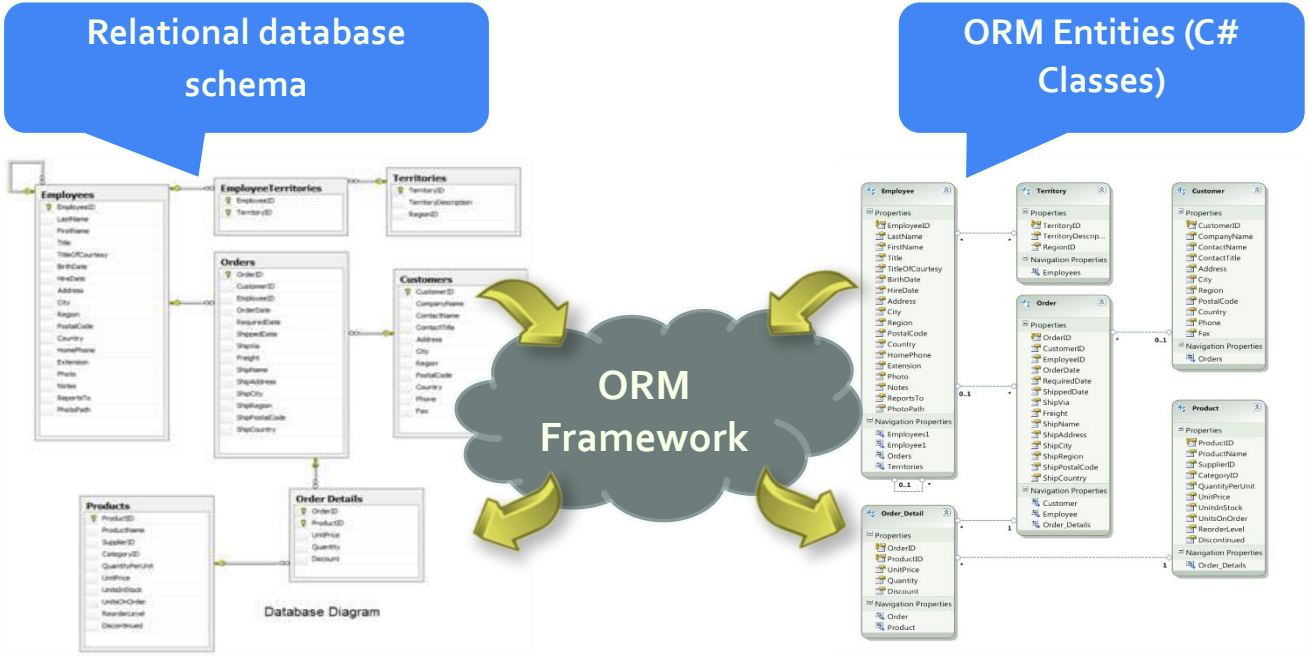
Introduction to ORM

- Object-Relational Mapping (ORM) is a programming technique for automatic mapping and converting data
- Between relational database tables and object-oriented classes and objects
- ORM creates a "virtual object database"
- ORM frameworks automate the ORM process

ORM Frameworks

- ORM frameworks typically provide the following functionality:
  - Creating object model by database schema (Database First)
  - Creating database schema by object model (Code First)
  - Querying data by object-oriented API
- Data manipulation operations
  - CRUD – create, retrieve, update, delete
- ORM frameworks automatically generate SQL to perform the requested data operations

ORM Mapping Example



ORM Advantages

- Developer productivity
  - Writing less code
- Abstract from differences between object and relational world
- Manageability of the CRUD operations for complex relationships
- Easier maintainability

Entity Framework (EF) Overview

- Provides a runtime infrastructure for managing SQL-based database data as .NET objects
- The relational database schema is mapped to an object model (classes and associations)
- Visual Studio has built-in tools for generating Entity Framework SQL data mappings

Entity Framework Features

- Maps tables, views, stored procedures and functions as .NET objects
- Provides LINQ-based data queries
- Executed as SQL SELECTs on the database server (parameterized queries)
- Built-in CRUD operations – Create/Read/Update/Delete
- Creating or deleting the database schema
- Tracks changes to in-memory objects

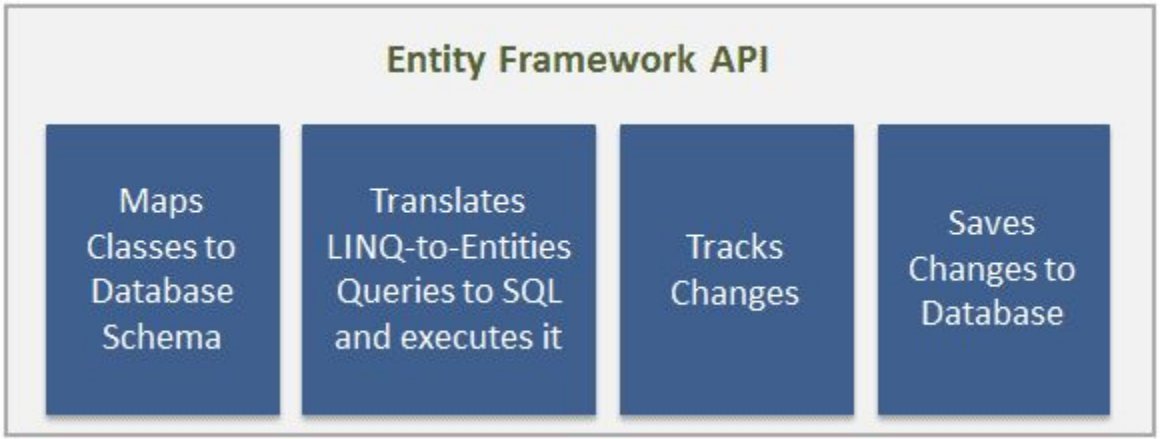
Entity Framework Components (1 of 2)

- The DbContext class
  - DbContext holds the database connection and the entity classes
  - Provides LINQ-based data access
  - Implements identity tracking, change tracking, and API for CRUD operations
- Entity classes
  - Each database table is typically mapped to a single entity class (C# class)

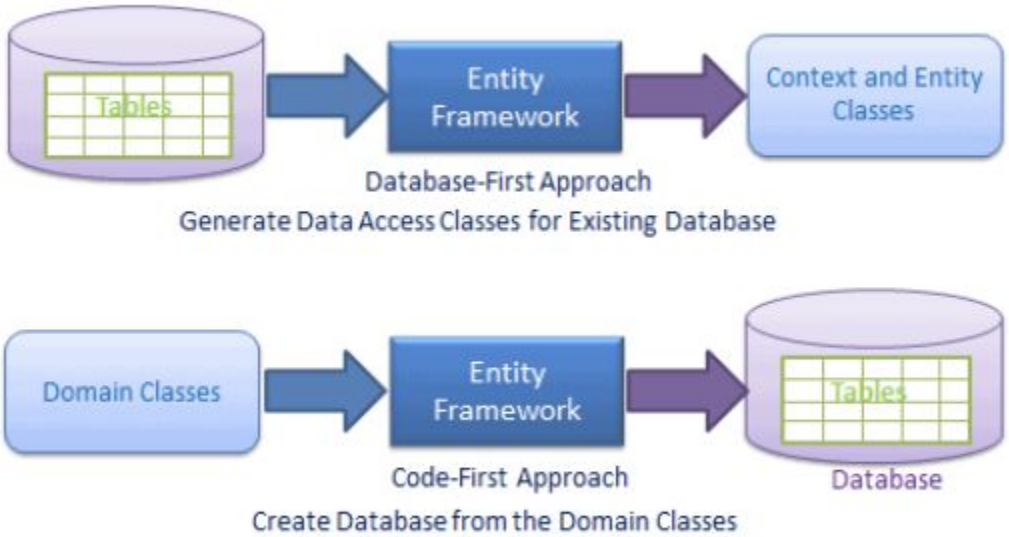
Entity Framework Components (2 of 2)

- Associations (Relationship Management)
  - An association is a primary key / foreign key based relationship between two entity classes
  - Allows navigation from one entity to another, e.g. Student. Courses
- Concurrency control
  - Entity Framework uses optimistic concurrency control (no locking by default)
  - Provides automatic concurrency conflict detection and means for conflicts resolution

Entity Framework API



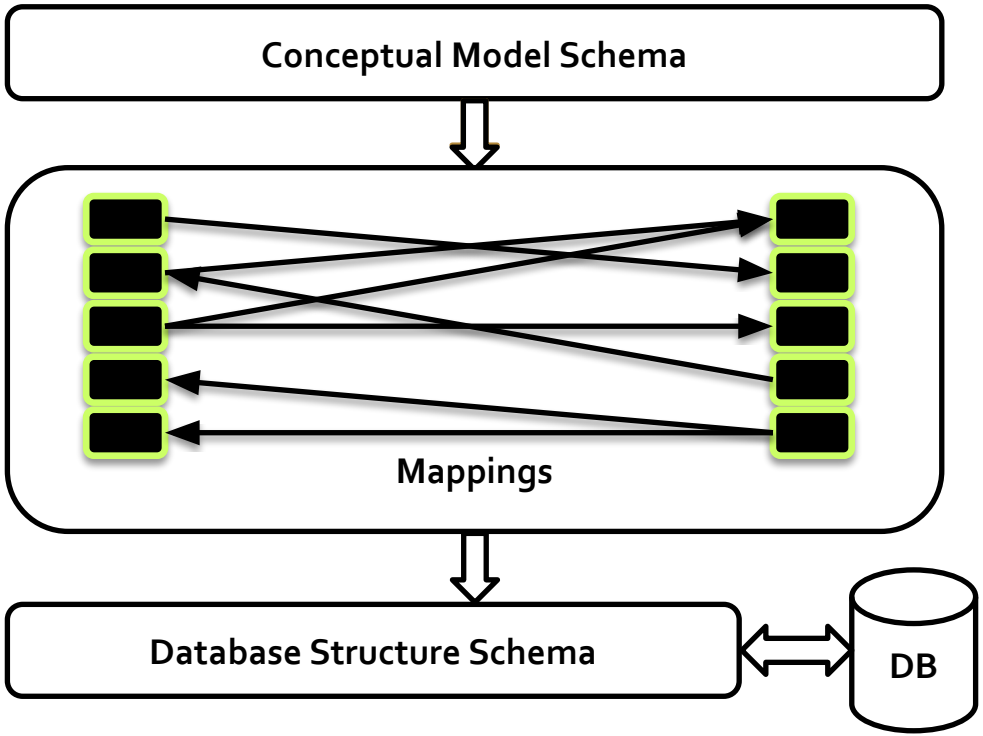
Entity Framework Core Workflows



Entity Framework Core Data Providers

Database	NuGet Package
SQL Server	Microsoft.EntityFrameworkCore.SqlServer
MySQL	MySql.Data.EntityFrameworkCore
PostgreSQL	Npgsql.EntityFrameworkCore.PostgreSQL
SQLite	Microsoft.EntityFrameworkCore.SQLite
SQL Compact	EntityFrameworkCore.SqlServerCompact40
In-memory	Microsoft.EntityFrameworkCore.InMemory

Entity Framework Runtime Metadata



Reading Data with LINQ Query (1 of 2)

```
using (var context = new NorthwindEntities())
{
   var customerPhoness = context.Customers
                              .Select(c => c.Phone)
                              .Where(c => c.City == "London")
                              .ToList();
}
```

This is called projection

ToList() method executes the query

Reading Data with LINQ Query (1 of 2)

- Find element by id

```
using (var context = new NorthwindEntities())
{
  var customer = context.Customers.Find(2);
  Console.WriteLine(customer.ContactTitle);
}
```

Create New Data

```
// Create new order object
Order order = new Order()
{
  OrderDate = DateTime.Now,
  ShipName = "Titanic",
  ShippedDate = new DateTime(1912, 4, 15),
  ShipCity = "Bottom Of The Ocean"
};
// Mark the object for inserting
context.Orders.Add(order);
context.SaveChanges();
```

This will execute an SQL INSERT

Cascading Inserts

```
Country spain = new Country();
spain.Name = "Spain";
spain.Population = "46 030 10";
spain.Cities.Add(new City { Name = "Barcelona"} );
spain.Cities.Add(new City { Name = "Madrid"} );
countryEntities.Countries.Add(spain);
countryEntities.SaveChanges();
```

Update Existing Data

- DbContext allows modifying entity properties and persisting them in the database
  - Load an entity, modify it and call SaveChanges()
- The DbContext automatically tracks all changes made on its entity objects

```
Order order = northwindEntities.Orders.First();
order.OrderDate = DateTime.Now;
context.SaveChanges();
```

This will execute an SQL UPDATE

This will execute an SQL SELECT to load the first order

Delete Existing Data

- Delete is done by Remove() on the specified entity collection
- SaveChanges() method performs the delete action in the database

```
Order order = northwindEntities.Orders.First();

// Mark the entity for deleting on the next save
northwindEntities.Orders.Remove(order);
northwindEntities.SaveChanges();
```

This will execute an SQL DELETE command

Executing Native SQL Queries (1 of 2)

- Executing a native SQL query in Entity Framework directly in its database store:

```
ctx.Database.SqlQuery<return-type>(native-SQL-query);
```

```
string query = "SELECT count(*) FROM dbo.Customers";
var queryResult = ctx.Database.SqlQuery<int>(query);
int customersCount = queryResult.FirstOrDefault();
```

Executing Native SQL Queries (2 of 2)

- Native SQL queries can also be parameterized:

```
NorthwindEntities context = new NorthwindEntities();

string nativeSQLQuery =
  "SELECT FirstName + ' ' + LastName " +
  "FROM dbo.Employees " +
  "WHERE Country = {0} AND City = {1}";
object[] parameters = { country, city };

var employees = context.Database.SqlQuery<string>(string.Format(nativeSQLQuery,
parameters));
foreach (var emp in employees)
{
  Console.WriteLine(emp);
}
```

Attaching and Detaching Objects

- In Entity Framework, objects can be attached to or detached from an object context
- Attached objects are tracked and managed by the DbContext
  - SaveChanges() persists all changes in DB
- Detached objects are not referenced by the DbContext
  - Behave like a normal objects, like all others, which are not related to EF

Attaching Detached Objects

- When a query is executed inside an DbContext, the returned objects are automatically attached to it
- When a context is destroyed, all objects in it are automatically detached
  - E.g. in Web applications between the requests
- You might later on attach to a new context objects that have been previously detached

Detaching Objects

- When an object is detached?
  - When we obtain the object from an DbContext and then Dispose it
  - Manually: by set the entry state to Detached

```
Product GetProduct(int id)
{
   using (NorthwindEntities northwindEntities = new NorthwindEntities())
   {
      return northwindEntities.Products.First(p => p.ProductID == id);
   }
}
```

Now the returned product is detached

Attaching Objects

- When we want to update a detached object we need to reattach it and then update it
  - Done by the Attached state of the entry

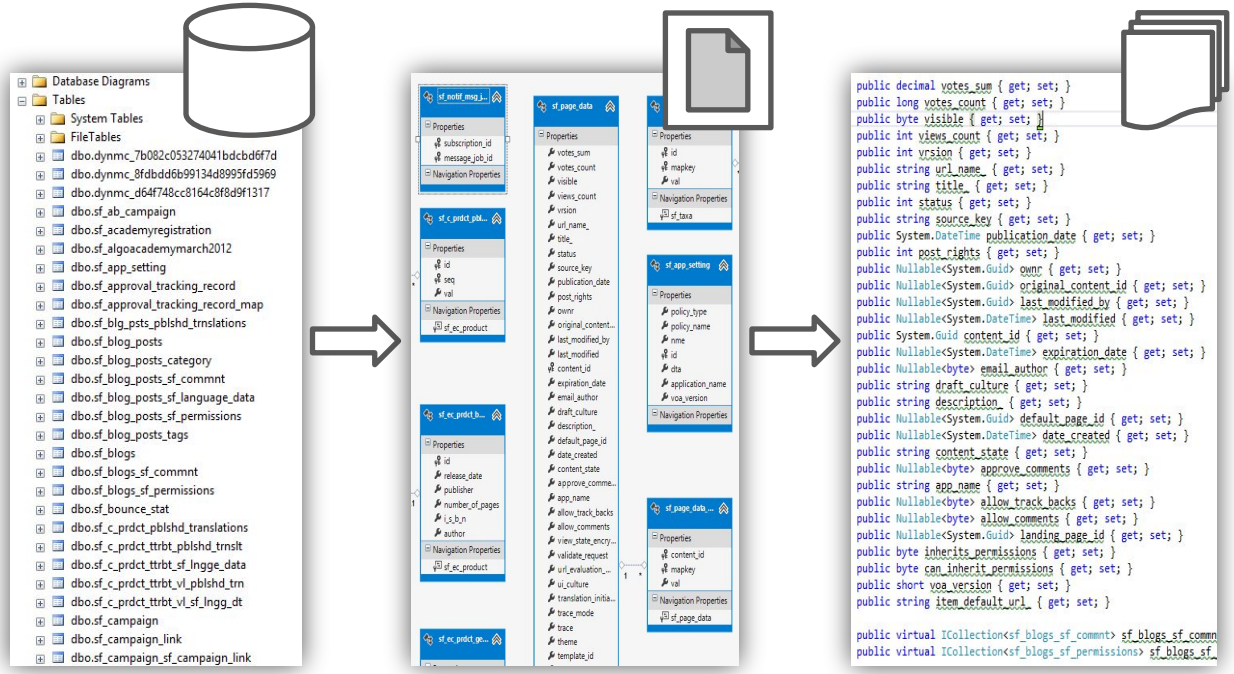
```
void UpdatePrice(Product product, decimal newPrice)
{
   using (NorthwindEntities northwindEntities = new NorthwindEntities())
   {
      var entry = northwindEntities.Entry(product);
      entry.State = EntityState.Attached;
      product.UnitPrice = newPrice;
      northwindEntities.SaveChanges();
   }
}
```

Modeling Workflow

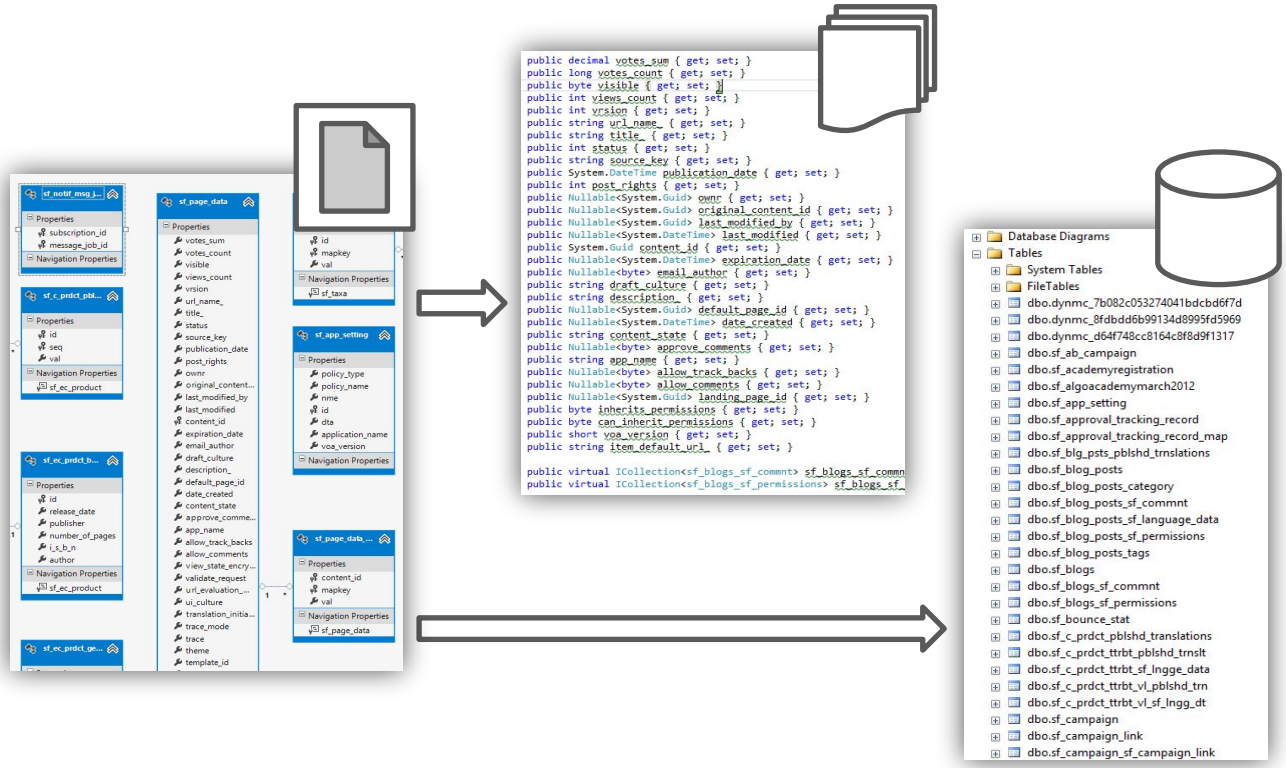
- **Database first**
  - Create models as database tables
  - Use Management Studio or native SQL queries
- **Model first**
  - Create models using visual EF designer in VS
- **Code first**
  - Write models and combine them in DbContext

Data First Modeling

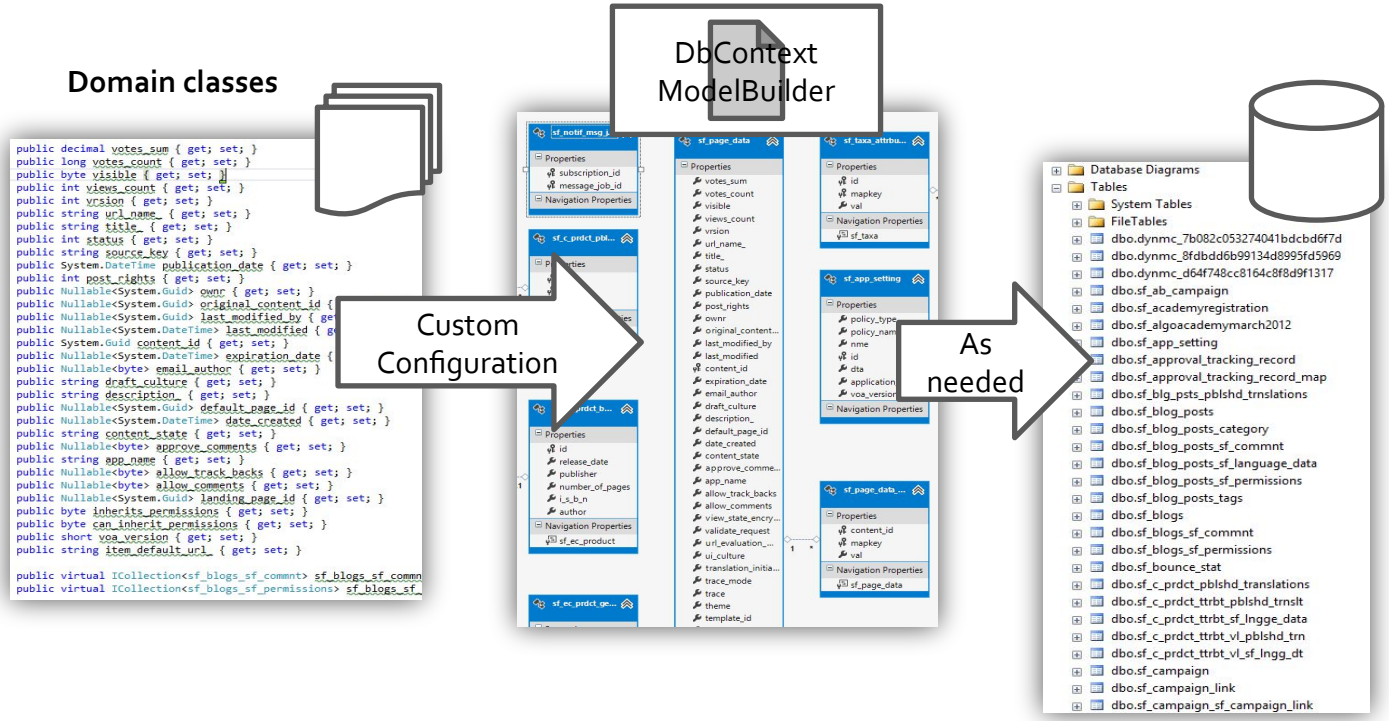
- Create models as database tables and then generate code (models) from them

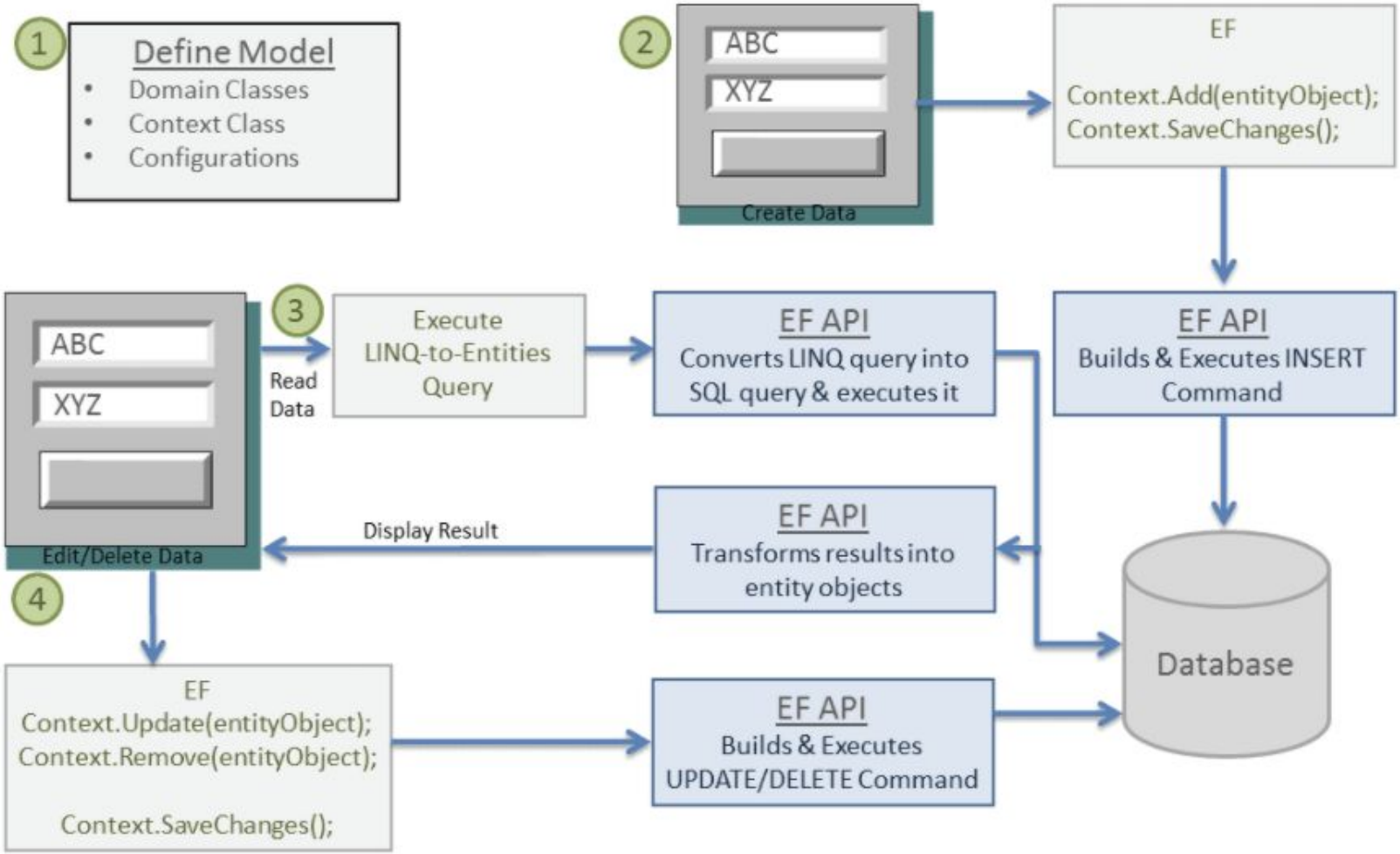


Model First Modeling



Code First Modeling




1
Define Model
Domain Classes
Context Class
Configurations
2
ABC
XYZ
Create Data
EF
Context.Add(entityObject);
Context.SaveChanges();
ABC
XYZ
Edit/Delete Data
3
Read Data
Execute LINQ-to-Entities Query
EF API
Converts LINQ query into SQL query & executes it
EF API
Builds & Executes INSERT Command
Display Result
EF API
Transforms results into entity objects
Database
4
EF
Context.Update(entityObject);
Context.Remove(entityObject);
Context.SaveChanges();
EF API
Builds & Executes UPDATE/DELETE Command

Why use Code First?

- Write code without having to define mappings in XML or create database models
- Define objects in POCO
  - Reuse these models and their attributes
- No base classes required
- Enables database persistence with no configuration
  - Can use automatic migrations
- Can use Data Annotations (Key, Required, etc.)

Domain Classes (Models/Entities) (1 of 2)

```
public class PostAnswer
{
    public int PostAnswerId { get; set; }
    public string Content { get; set; }
    public int PostId { get; set; }
    public virtual Post Post { get; set; }
}
```

Foreign key

Virtual for lazy loading

Navigation property

Domain Classes (Models/Entities) (2 of 2)

```
public class Post
{
    private ICollection<PostAnswer> answers;
    public Post()
    {
        this.answers = new HashSet<PostAnswer>();
    }
    // ...
    public virtual ICollection<PostAnswer> Answers
    {
        get { return this.answers; }
        set { this.answers = value; }
    }
    public PostType Type { get; set; }
}
```

Prevents null reference exception

Navigation property

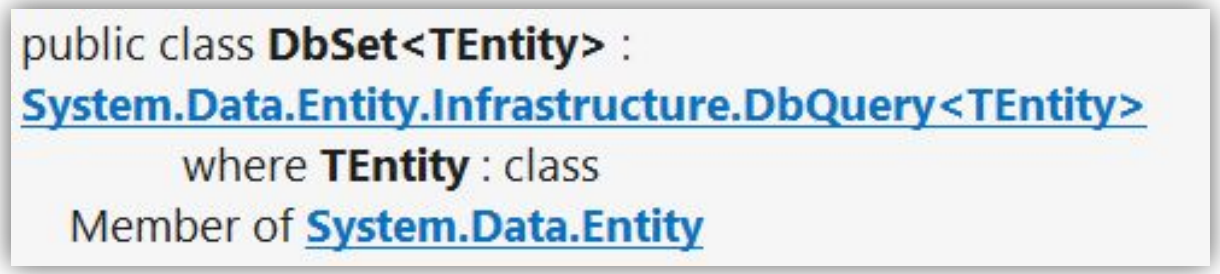
Enumeration

DbContext Class

- A class that inherits from DbContext
  - Manages model classes using DbSet type
  - Implements identity tracking, change tracking, and API for CRUD operations
  - Provides LINQ-based data access
- Recommended to be in a separate class library
  - Don't forget to reference the Entity Framework library (using NuGet package manager)

DbSet Type

- Collection of single entity type
- Set operations: Add, Attach, Remove, Find
- Use with DbContext to query database



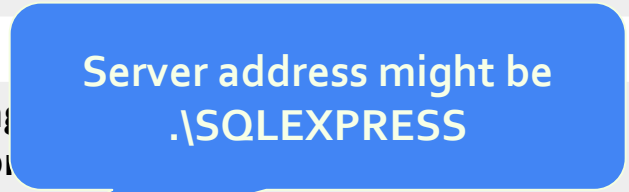

```
public DbSet<Post> Posts { get; set; }
```

DbContext Example

```
using System.Data.Entity;

using CodeFirst.Models;

public class ForumContext : DbContext
{
    public DbSet<Category> Categories { get; set; }

    public DbSet<Post> Posts { get; set; }

    public DbSet<PostAnswer> PostAnswers { get; set; }

    public DbSet<Tag> Tags { get; set; }
}
```

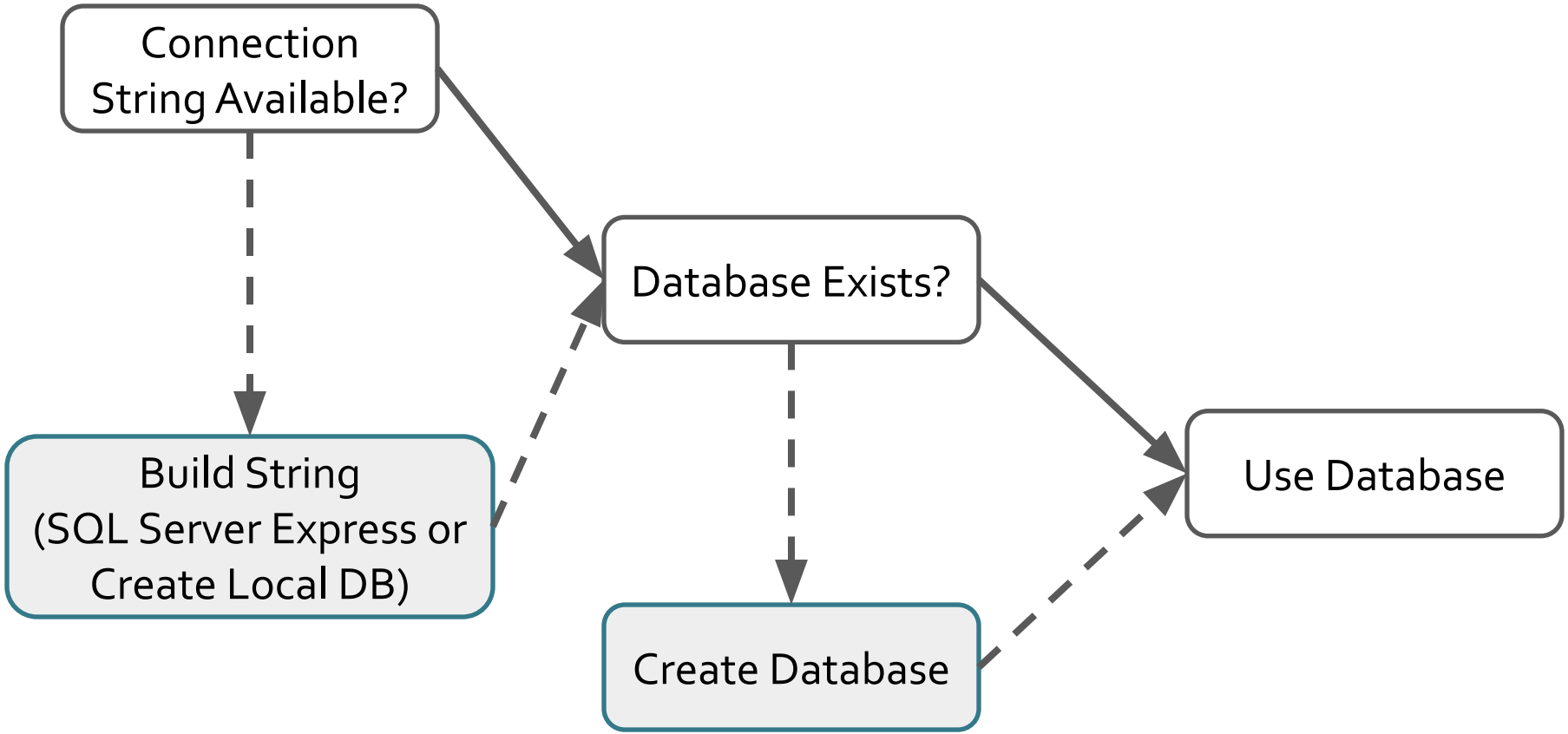
SQL Server Connection

```
public class ForumContext : DbContext
{
    public ForumContext()
        : base("ForumDb")
    { }
    // ...
}
```

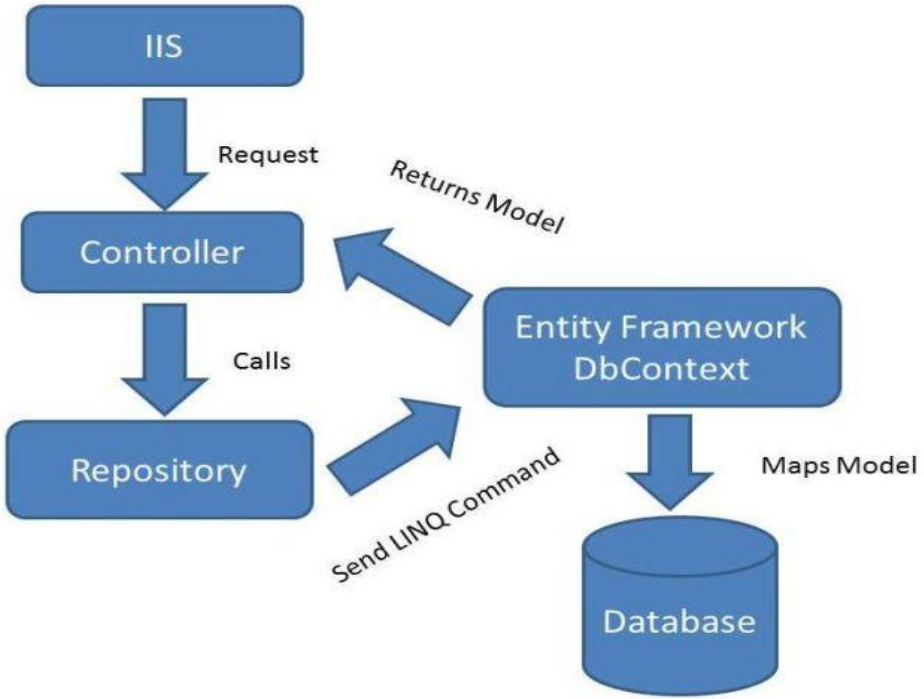



```
<connectionStrin
    <add name="Fo
        connectionSt      Data
        Source=.;Initial Catalog=ForumDb;Integrated Security=True"
        providerName="System.Data.SqlClient" />
</connectionStrings>
```

Database Connection Workflow



Repository Pattern (1 of 2)

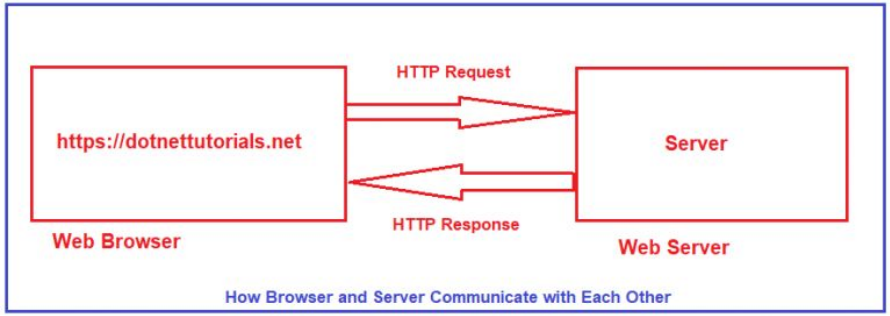


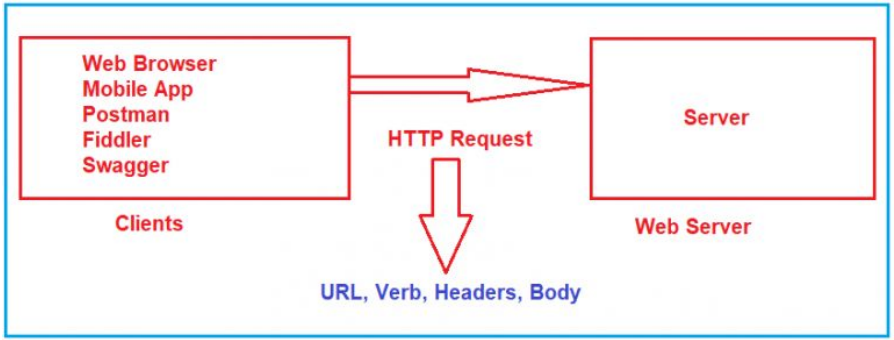
Repository Pattern (2 of 2)

- Gives abstraction over the data layer
- Single place to make changes to your data access
- Single place responsible for a set of tables
- Easily replaceable by other implementations
- Hides the details in accessing data
- Can be implemented in various ways

ASP .NET CORE

HTTP


HTTP Request
https://dotnettutorials.net
Server
HTTP Response
Web Browser
Web Server
How Browser and Server Communicate with Each Other


Web Browser
Mobile App
Postman
Fiddler
Swagger
Server
HTTP Request
Clients
Web Server
URL, Verb, Headers, Body

Introduction to Microsoft Web Technologies

Develop	Host	Execute	
		Server-Side	Client-Side
• Visual Studio • Visual Studio Code	• IIS • Microsoft Azure	• ASP.NET Core • ASP.NET 4.x	• JavaScript • jQuery • Angular • React • AJAX

Overview of ASP .NET

Programming Models

- ASP.NET 4.x
  - Web Pages
  - Web Forms
- ASP.NET Core
  - MVC
  - Web API
  - Razor Pages

ASP.NET Core features

Hosting

- Kestrel, Startup

Middleware

- Routing, authentication, static files, diagnostics, error handling, session, CORS, localization, custom

Dependency Injection

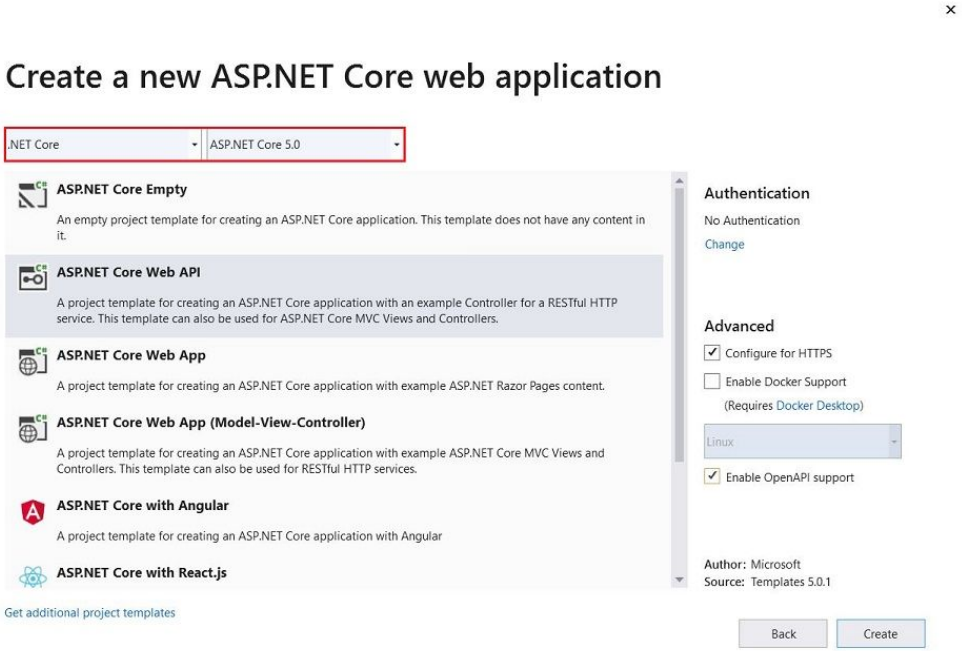
Configuration

Logging

Application frameworks

- Razor Pages, MVC, Identity, SignalR, Blazor, Workers, gRPC

Visual Studio ASP.NET Core Web API



Setup

Visual Studio 2019

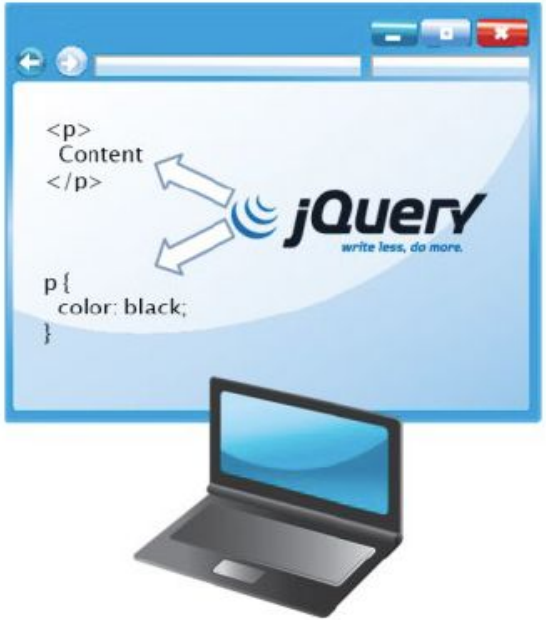
SQL Server LocalDb (installs with VS 2019)

.NET 5.0 SDK

You'll add EF Core 5.0 directly to the projects you create

Client-Side Technologies

- JavaScript
- jQuery
- AJAX
- Angular
- React
- And more

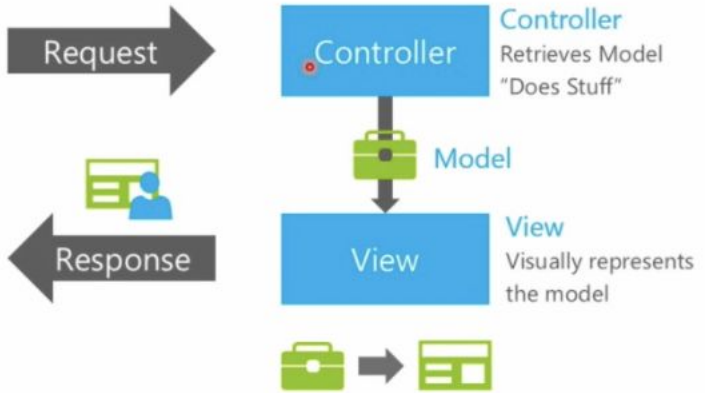


Hosting Technologies

- IIS
  - Features
  - Scaling
  - Perimeter Networks
- IIS Express
- Other Web Servers

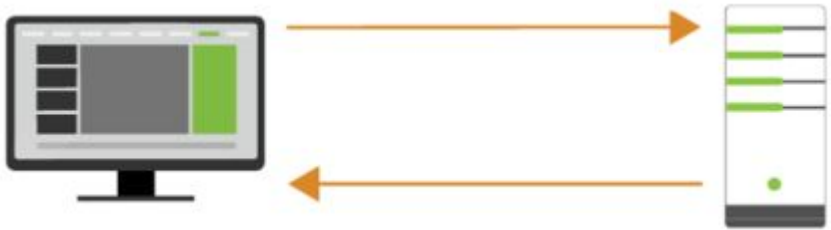
Overview of MVC

- Models encapsulate objects and data
- Views generated the user interface
- Controllers interact with user actions
- Code in .cshtml and .cs files
- Precise control of HTML and URLs
- Easy to use unit tests



Overview of Web API

- Helps creating RESTful APIs
- Enables external systems to use your application’s business logic
- Accessible to various HTTP clients
- Helps to obtain data in different formats such as JSON and XML
- It supports create, read, update, and delete (CRUD)
- Ideal for mobile application integration



Disadvantages of Web API

- With Web API you have a complete separation between server-side code and client-side code. For some programmers, this is an advantage, but others may find this challenging and hard to understand.

- The data returned from Web API usually is not indexed by search engines such as Google, more work needs to be done to make it crawlable.

Choose between .NET Core and .NET Framework

- You should use .NET Core when:
  - You want your code to run cross-platform
  - You want to create microservices
  - You want to use Docker containers
  - You want to achieve a high-preforming scalable system
- You should use .NET Framework when:
  - You want to extend an existing application that uses .NET Framework
  - You want to use NuGet packages or third-party .NET libraries that are not supported in .NET Core
  - You want to use .NET technologies that aren't supported in .NET Core
  - You want to use a platform that doesn't support .NET Core

Project Development Methodologies

- Waterfall Model
- Iterative Development Model
- Prototype Model
- Agile Development Model
- Extreme Programming
- Test Driven Development
- Unified Modeling Language

Gathering Requirements

- Functional requirements

  User interface requirements.

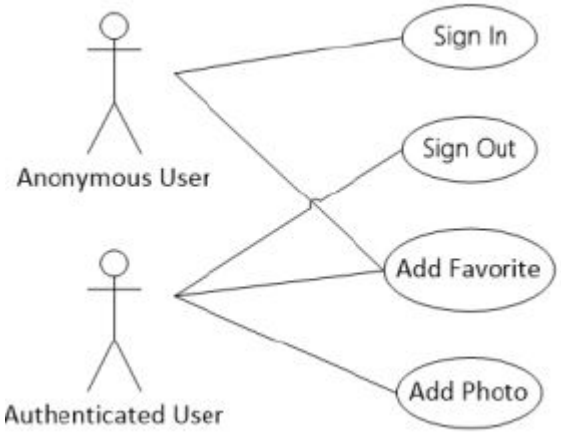
  They describe how the user interacts with an application.

  Usage requirements

  They describe what a user can do with the application.

  Business requirements

  They describe how the application will fulfill business functions.



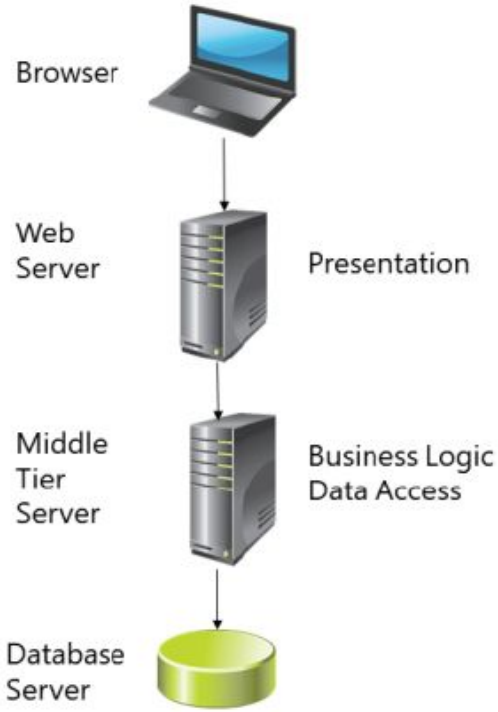
- Technical requirements (Non-functional)

Planning the Database Design

- Logical modeling
- Physical database structure
- Working with DBAs
- Database design in agile and extreme programming

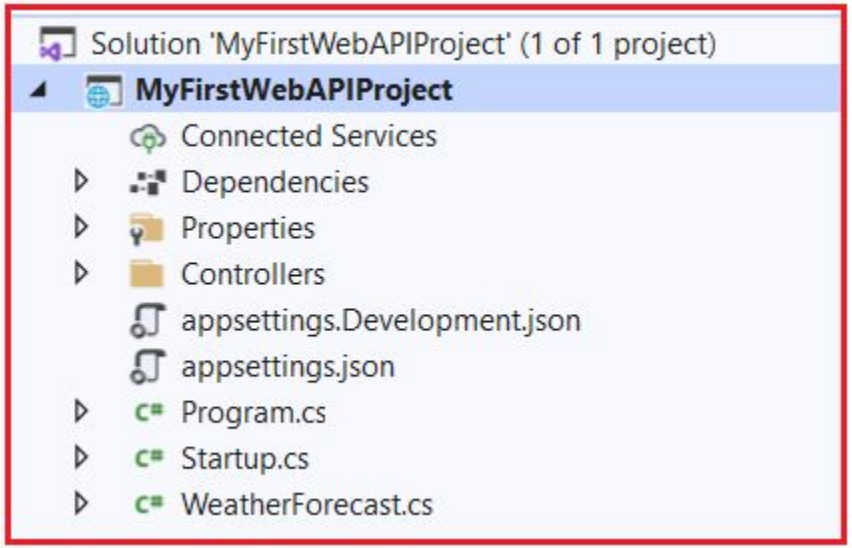
Planning for Distributed Applications

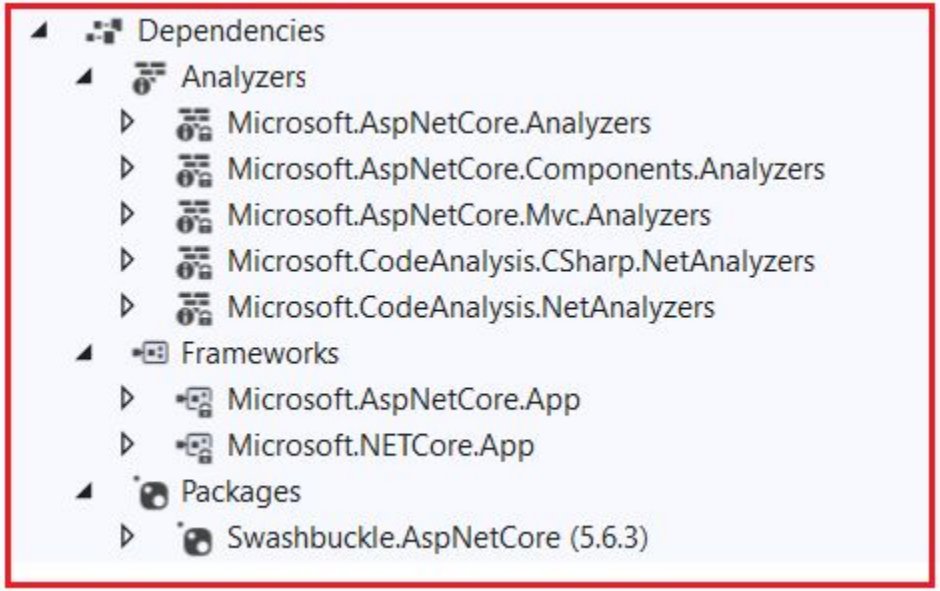
- Layers
  - Presentation
  - Business logic
  - Data access
  - Database
- Communication
- Security



ASP.NET Core Overview

ASP .NET Core Web API Files and Folders





The Startup class (1 of 2)

- Services required by the app are configured.
- The app's request handling pipeline is defined, as a series of middleware components.

The Startup class (2 of 2)

```
public class Startup {
  public void ConfigureServices(IServiceCollection services) {
    services.AddControllers();
    services.AddDbContext < TodoContext > (opt => opt.UseInMemoryDatabase("TodoList"));
  }

  public void Configure(IApplicationBuilder app) {
    if (env.IsDevelopment()) {
      app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
      endpoints.MapControllers();
    });
  }
}
```

Dependency injection (services)

- ASP.NET Core includes a built-in dependency injection (DI) framework that makes configured services available throughout an app.
- For example, a logging component is a service.

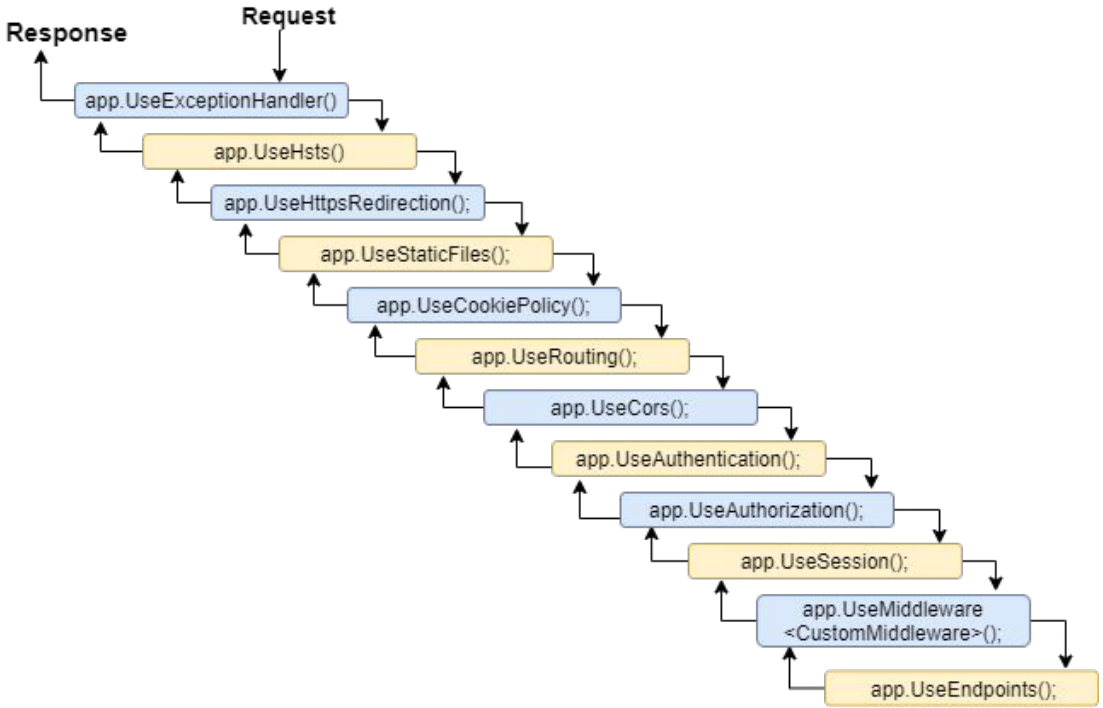
```
public class Startup {

  public void ConfigureServices(IServiceCollection services) {
    services.AddControllers();
    services.AddDbContext < TodoContext > (opt => opt.UseInMemoryDatabase("TodoList"));
  }

}
```

Middleware

- The request handling pipeline is composed as a series of middleware components.
- Each component performs operations on an HttpContext and either invokes the next middleware in the pipeline or terminates the request.

```
  public void Configure(IApplicationBuilder app) {
    if (env.IsDevelopment()) {
      app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
      endpoints.MapControllers();
    });
  }
}
```

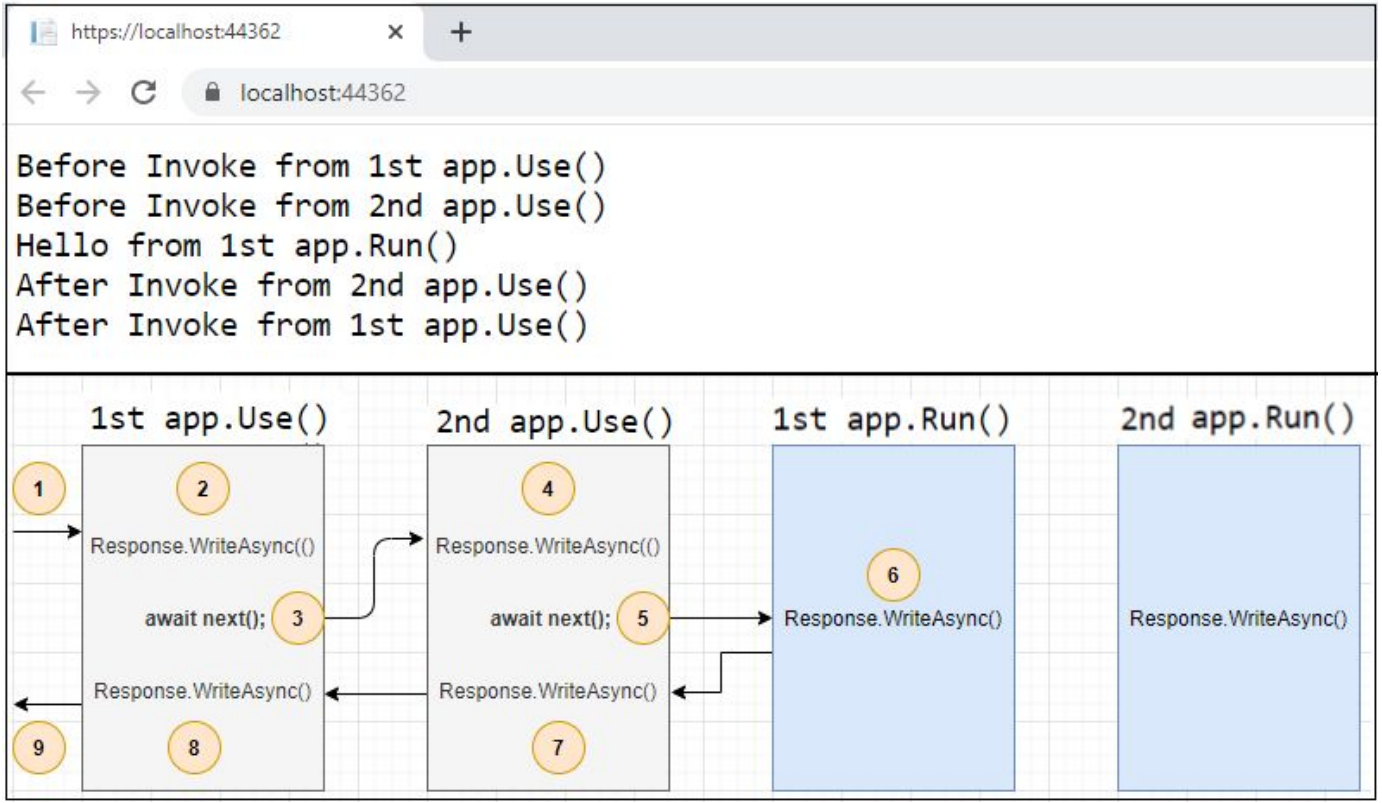
Middleware Suggested Order



Understanding the Run, Use and Map Method (1 of 3)

- app.Run()
  - This middleware component may expose Run[Middleware] methods that are executed at the end of the pipeline.
  - Generally, this acts as a terminal middleware and is added at the end of the request pipeline, as it cannot call the next middleware.
- app.Use()
  - This is used to configure multiple middleware.
  - Can call the next request delegate in the pipeline.
  - Can also short-circuit (terminate) the pipeline by not calling the next parameter.

Understanding the Run, Use and Map Method (2 of 3)



Understanding the Run, Use and Map Method (3 of 3)

- app.Map()
  - These extensions are used as a convention for branching the pipeline.
  - The map branches the request pipeline based on matches of the given request path.
  - If the request path starts with the given path, the branch is executed.

Host (1 of 3)

- On startup, an ASP.NET Core app builds a host.
- The host encapsulates all of the app's resources, such as:
  - An HTTP server implementation
  - Middleware components
  - Logging
  - Dependency injection (DI) services
  - Configuration
- There are two different hosts:
  - .NET Generic Host
  - ASP.NET Core Web Host

Host (2 of 3)

- The .NET Generic Host is recommended. The ASP.NET Core Web Host is available only for backwards compatibility.

```
public class Program {
  public static void Main(string[] args) {
    CreateHostBuilder(args).Build().Run();
  }

  public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
    .ConfigureWebHostDefaults(webBuilder => {
      webBuilder.UseStartup < Startup > ();
    });
}
```

Host (2 of 3)

- The CreateDefaultBuilder and ConfigureWebHostDefaults methods configure a host with a set of default options, such as:
  - Use **Kestrel** as the web server and enable IIS integration.
  - Load configuration from appsettings.json, appsettings.{Environment Name}.json, environment variables, command line arguments, and other configuration sources.
  - Send logging output to the console and debug providers.

Configuration

- ASP.NET Core provides a configuration framework
- Settings as name-value pairs from an ordered set of configuration providers
- Built-in configuration providers
  - JSON files
  - XML files
  - Command-line arguments
- ASP.NET Core apps are configured to read from appsettings.json by default

Environments (1 of 2)

- Execution environments, such as Development, Staging, and Production, are a first-class notion in ASP.NET Core
- Specify the environment an app is running in by setting the ASPNETCORE_ENVIRONMENT environment variable.

Environments (2 of 2)

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
  if (env.IsDevelopment()) {
    app.UseDeveloperExceptionPage();
  }

  app.UseHttpsRedirection();

  app.UseRouting();

  app.UseAuthorization();

  app.UseEndpoints(endpoints => {
    endpoints.MapControllers();
  });
}
```

Logging (1 of 2)

- ASP.NET Core supports a logging API
- Available built-in and third-party logging providers
  - Console
  - Debug
  - Event Tracing on Windows
  - Windows Event Log
  - TraceSource
  - Azure App Service
  - Azure Application Insights

Logging (2 of 2)

```
public class TodoController: ControllerBase {
  private readonly ILogger _logger;

  public TodoController(ILogger < TodoController > logger) {
    _logger = logger;
  }

  [HttpGet("{id}", Name = "GetTodo")]
  public ActionResult < TodoItem > GetById(string id) {
    _logger.LogInformation(LoggingEvents.GetItem, "Getting item {Id}", id);

    // Item lookup code removed.

    if (item == null) {
      _logger.LogWarning(LoggingEvents.GetItemNotFound, "GetById({Id}) NOT FOUND", id);
      return NotFound();
    }

    return item;
  }
}
```

Routing

- A route is a URL pattern that is mapped to a handler.
- The handler is typically a Razor page, an action method in an MVC controller, or a middleware.
- ASP.NET Core routing gives you control over the URLs used by your app.

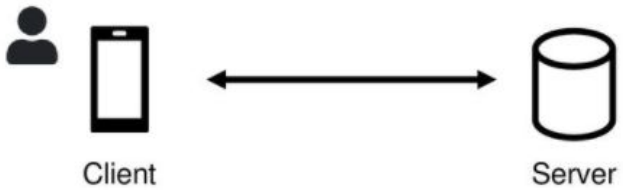
Error Handling

- ASP.NET Core has built-in features for handling errors, such as:
  - A developer exception page
  - Custom error pages
  - Static status code pages
  - Startup exception handling

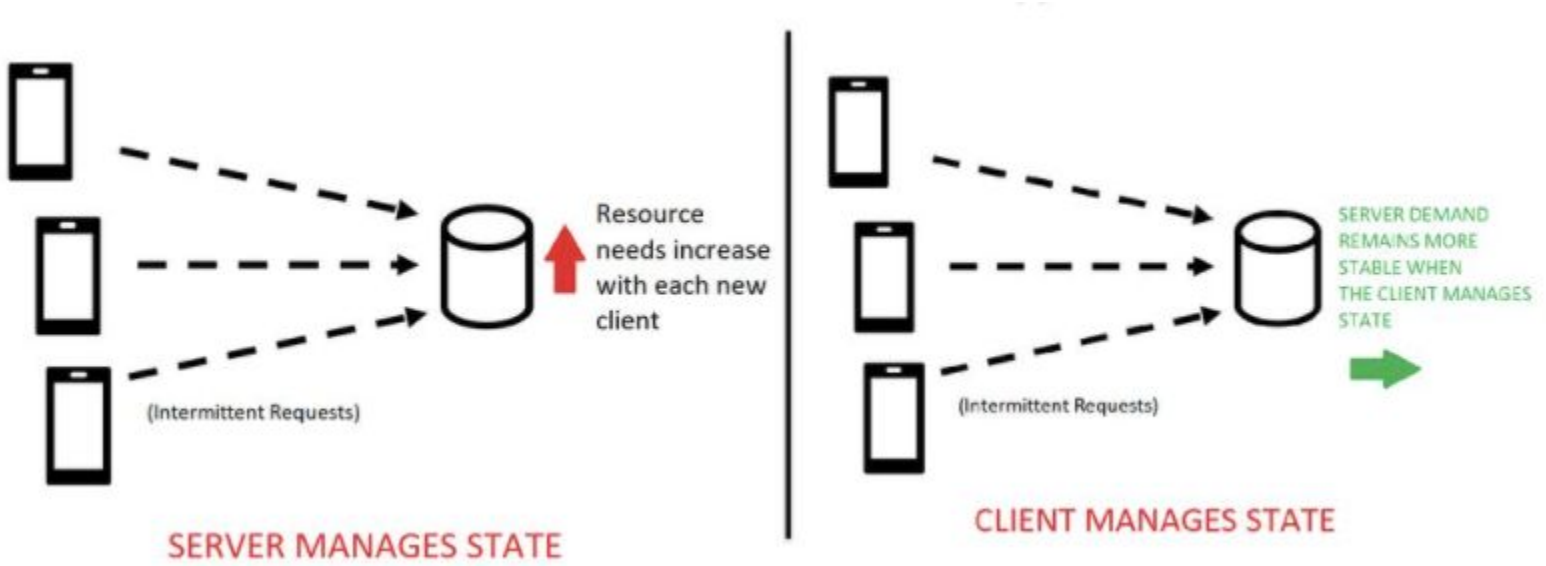
RESTful API Introduction

What is REST?

- REpresentational State Transfer" (REST) First described by Roy Fielding (in his dissertation)
- REST is a set of principles that define how Web standards, such as HTTP and URIs, are supposed to be used
- REST does NOT propose an alternative to the web (like SOAP/WS-*, CORBA, RMI)
- REST provides a common and consistent interface based on proper use of HTTP



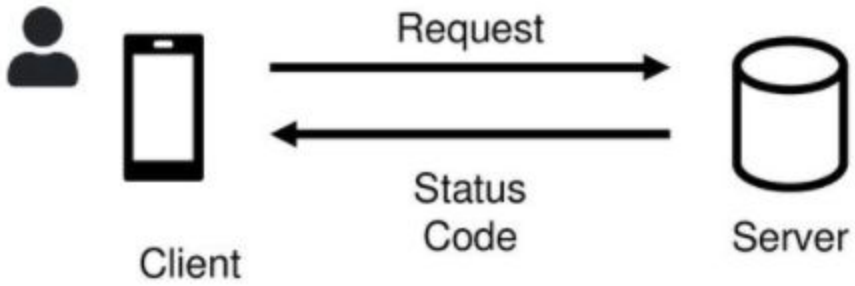
State


Resource needs increase with each new client
(Intermittent Requests)
SERVER MANAGES STATE
SERVER DEMAND REMAINS MORE STABLE WHEN THE CLIENT MANAGES STATE
(Intermittent Requests)
CLIENT MANAGES STATE

Use of Standard HTTP Methods

HTTP action /URL		Meaning
GET /things	->	Retrieve *list* of things
GET /things/23	->	Retrieve *one* thing identified with 23
POST /things	->	Create a new thing
PATCH /things/23	->	Update thing 23
PUT /things/23	->	Replace thing 23 (or create thing 23)*
DELETE /things/23	->	Delete thing 23

RESTful API Conventions - HTTP Status Codes



Standard HTTP Status Codes (1 of 2)

Status Code	Description
200	The request was successful.
201	A resource was successfully created.
202	Request has been received.
304	ETag matches 'If-None-Match' header - not changed
400	The request cannot be understood.
401	Authentication failed, or not authorized.
404	The requested resource could not be found.
405	The method is not supported by the resource.

Standard HTTP Status Codes (2 of 2)

Status Code	Description
406	Not Acceptable. Accept header identifies an unsupported format.
409	There is a conflict (that violates an optimistic lock).
412	A precondition failed. Used for conditional requests.
415	Unsupported Media Type. Client requested an unsupported format.
417	Expectation Failed.
422	'Unprocessable' entity.
429	The rate limit is exceeded
500	An undisclosed server error occurred. This is a generic 'fail whale' response.

Resource

- A resource is anything that's important enough to be referenced as a thing in itself.
- Thinking about solutions in terms of resources
- REST principles have been shown to reduce complexity
- 'Resources' often correspond to persistent domain objects

Common Constraints

- Abuse of GET method

  Get is assumed to be a safe operation that never modifies that state of the resource. But it has been often contorted into performing other functions.

  http://api.example.com?action=ADD&product=Phone

- APIs are not self documenting
- State matters

  The burden of managing state is in the client. There is no one consistent approach to managing state.

URIs and Resources (1 of 2)

- URI is an ‘address’ of a resource
- A resource must have *at least one* URI
- No URI Not a resource URIs should be descriptive (human parseable) and have structure. For Example:
  - http://www.ex.com/software/releases/latest.tar.gz
  - http://www.ex.com/map/roads/USA/CA/17_mile_drive
  - http://www.ex.com/search/cs578
  - http://www.ex.com/sales/2012/Q1
  - http://www.ex.com/relationships/Alice;Bob

URIs and Resources (2 of 2)

- Not so good URIs (everything as query parameters):
  - http://www.ex.com?**software**=Prism&**release**=latest&**filetype**=tar&**method**=fetch
  - http://www.ex.com?**sessionId**=12345678908765432123456787654323456786543234567887654 3&**itemId**=9AXFE5&**method**=addToCart
- URIs need not have structure/predictability
- Each URI must refer a unique resource – although they may point to the ‘same one’ at some point in time (Ex.: …/latest.tar.gz and …/v1.5.6.tar.gz)

REST Principle #1

The key abstraction of information is a resource, named by a URI. Any information that can be named can be a resource

REST - Statelessness

- Every HTTP request happens in complete isolation
  - Server NEVER relies on information from prior requests
  - There is no specific ‘ordering’ of client requests (i.e. page 2 may be requested before page 1)
- If the server restarts a client can resend the request and continue from it left off
- *Possible states* of a server are also resources and should be given their own URIs!

REST Principle #2

All interactions are context-free: each interaction contains all of the information necessary to understand the request, independent of any requests that may have preceded it.

REST - Representations

- Resources are NOT data – they are an abstraction of how the information/data is split up for presentation/consumption
- The web server must respond to a request by sending a series of bytes in a specific file format, in a specific language
  - i.e. a *representation* of the resource
  - Formats: XML/JSON, HTML, PDF, PPT, DOCX…
  - Languages: English, Spanish, Hindi, Portuguese…

Which Representation to Request?

- Style 1: Distinct URI for each representation:
  - ex.com/press-release/2012-11.en (English)
  - ex.com/press-release/2012.11.fr (French)
- Style 2: Content Negotiation
  - Expose Platonic form URI: ex.com/press-release/2012-11
  - Client sets specific HTTP request headers to signal what representations it's willing to accept
    - **Accept:** Acceptable file formats
    - **Accept-Language:** Preferred language

REST Principle #3

The representation of a resource is a sequence of bytes, plus representation metadata to describe those bytes. The particular form of the representation can be negotiated between REST components

REST - Uniform Interface

- HTTP Provides 4 basic methods for CRUD (create, read, update, delete) operations:
  - **GET**: Retrieve representation of resource
  - **PUT**: Update/modify existing resource (or create a new resource)
  - **POST**: Create a new resource
  - **DELETE**: Delete an existing resource
- Less commonly used methods:
  - **HEAD**: Fetch meta-data of representation only
  - **OPTIONS**: Check which HTTP methods a particular resource supports

HTTP Request/Response

Method	Request Entity-Body/Representation	Response Entity-Body/Representation
GET	(Usually) Empty Representation/entity-body sent by client	Server returns representation of resource in HTTP Response
DELETE	(Usually) Empty Representation/entity-body sent by client	Server may return entity body with status message or nothing at all
PUT	(Usually) Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all
POST	Client's proposed representation of resource in entity-body	Server may respond back with status message or with copy of representation or nothing at all

Steps to a RESTful Architecture

Read the Requirements and turn them into resources

- Figure out the data set
- Split the data set into resources

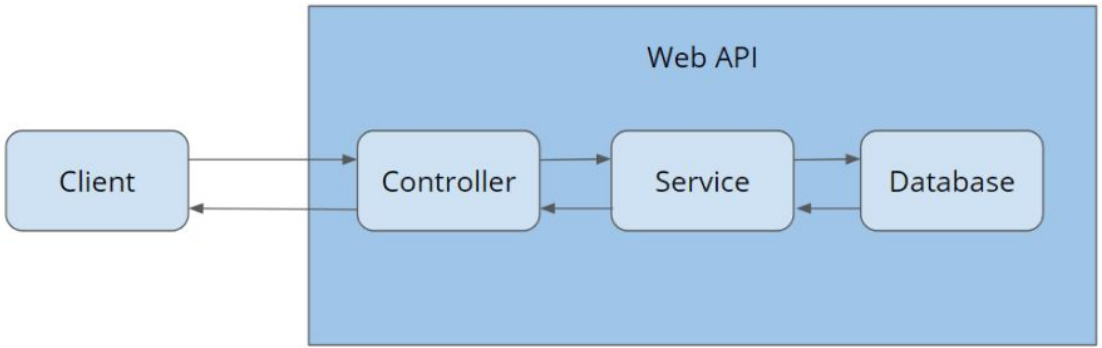
For each kind of resource:

- Name resources with URIs
- Expose a subset of uniform interface
- Design representation(s) accepted from client (e.g. JSON)
- Design representation(s) served to client (file-format, language and/or (which) status message to be sent)
- Consider typical course of events: sunny-day scenarios
- Consider alternative/error conditions: rainy-day scenarios

Web API with ASP.NET Core

ASP.NET Core Web API

- ASP.NET Core supports creating RESTful services, also known as web APIs, using C#
- Web API uses Controllers that derive from ControllerBase for handling web API requests
- A web API consists of one or more controller classes that derive from ControllerBase



When to use ControllerBase class?

- Don't create a web API controller by deriving from the Controller class.
- Controller derives from ControllerBase and adds support for views, it's for handling web pages, not web API requests.
- There's an exception to this rule: if you plan to use the same controller for both views and web APIs, derive it from Controller.

ControllerBase class

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

ControllerBase Properties and Methods (1 of 2)

- The ControllerBase class provides many properties and methods that are useful for handling HTTP requests

```
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult < Pet > Create(Pet pet) {

  pet.Id = _petsInMemoryStore.Any() ? _petsInMemoryStore.Max(p => p.Id) + 1 : 1;
  _petsInMemoryStore.Add(pet);

  return CreatedAtAction(nameof(GetById), new {
    id = pet.Id
  }, pet);

}
```

ControllerBase Properties and Methods (2 of 2)

Method	Notes
BadRequest	Returns 400 status code.
NotFound	Returns 404 status code.
PhysicalFile	Returns a file.
TryUpdateModelAsync	Invokes model binding.
TryValidateModel	Invokes model validation.

Attributes

- The **Microsoft.AspNetCore.Mvc** namespace provides attributes that can be used to configure the behavior of web API controllers and action methods.

Attribute	Notes
[Route]	Specifies URL pattern for a controller or action.
[Bind]	Specifies prefix and properties to include for model binding.
[HttpGet]	Identifies an action that supports the HTTP GET action verb.
[Consumes]	Specifies data types that an action accepts.
[Produces]	Specifies data types that an action returns.

Attribute on specific controllers

- The **[ApiController]** attribute can be applied to specific controllers

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Attribute on multiple controllers

```
[ApiController]
public class MyControllerBase : ControllerBase
{
}
```

```
[Produces(MediaTypeNames.Application.Json)]
[Route("[controller]")]
public class PetsController : MyControllerBase
```

Attribute routing requirement

- The [ApiController] attribute makes attribute routing a requirement

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
```

Binding source parameter inference

- A binding source attribute defines the location at which an action parameter's value is found

Attribute	Binding source
[FromBody]	Request body
[FromForm]	Form data in the request body
[FromHeader]	Request header
[FromQuery]	Request query string parameter
[FromRoute]	Route data from the current request
[FromServices]	The request service injected as an action parameter

Example - Binding source attribute

```
[HttpGet]
public ActionResult<List<Product>> Get(
    [FromQuery] bool discontinuedOnly = false)
{
    List<Product> products = null;

    if (discontinuedOnly)
    {
        products = _productsInMemoryStore.Where(p => p.IsDiscontinued).ToList();
    }
    else
    {
        products = _productsInMemoryStore;
    }

    return products;
}
```

[Consumes] attribute

- By default, an action supports all available request content types
- The [Consumes] attribute allows an action to limit the supported request content types

```
[HttpPost]
[Consumes("application/json")]
public IActionResult CreateProduct(Product product)
```

[Consumes] attribute

- The [Consumes] attribute also allows an action to influence its selection based on an incoming request's content type by applying a type constraint

```
[ApiController]
[Route("api/[controller]")]
public class ConsumesController: ControllerBase {
  [HttpPost]
  [Consumes("application/json")]
  public IActionResult PostJson(IEnumerable < int > values) =>
    Ok(new {
      Consumes = "application/json", Values = values
    });

  [HttpPost]
  [Consumes("application/x-www-form-urlencoded")]
  public IActionResult PostForm([FromForm] IEnumerable < int > values) =>
    Ok(new {
      Consumes = "application/x-www-form-urlencoded", Values = values
    });
}
```

Controller action return types

- ASP.NET Core offers the following options for web API controller action return types:
  - Specific type
  - IActionResult
  - ActionResult<T>

Specific Type

- The simplest action returns a primitive or complex data type (for example, string or a custom object type).

```
[HttpGet]
public List<Product> Get() =>
    _repository.GetProducts();
```

Return IEnumerable<T>

- In ASP.NET Core 2.2 and earlier, returning IEnumerable<T> from an action results in synchronous collection iteration by the serializer.
- The result is the blocking of calls and a potential for thread pool starvation.

```
public IEnumerable<Product> GetOnSaleProducts() =>
    _context.Products.Where(p => p.IsOnSale);
```

Return IEnumerable<T>

- To avoid synchronous enumeration and blocking waits on the database in ASP.NET Core 2.2 and earlier, invoke ToListAsync:

```
public async Task<IEnumerable<Product>> GetOnSaleProducts() =>
    await _context.Products.Where(p => p.IsOnSale).ToListAsync();
```

IAsyncEnumerable<T>

- In ASP.NET Core 3.0 and later, returning IAsyncEnumerable<T> from an action:
  - No longer results in synchronous iteration.
  - Becomes as efficient as returning IEnumerable<T>.

```
[HttpGet("asyncsale")]
public async IAsyncEnumerable<Product> GetOnSaleProductsAsync()
{
    var products = _repository.GetProductsAsync();

    await foreach (var product in products)
    {
        if (product.IsOnSale)
        {
            yield return product;
        }
    }
}
```

IActionResult type

- The IActionResult return type is appropriate when multiple ActionResult return types are possible in an action.
- The ActionResult types represent various HTTP status codes
- Some common return types in this category are BadRequestResult (400), NotFoundResult (404), and OkObjectResult (200).

Synchronous action

```
[HttpGet("{id}")]
[ProducesResponseType(StatusCodes.Status200OK, Type = typeof (Product))]
[ProducesResponseType(StatusCodes.Status404NotFound)]
public IActionResult GetById(int id) {
  if (!_repository.TryGetProduct(id, out
      var product)) {
    return NotFound();
  }

  return Ok(product);
}
```

Asynchronous action

```
[HttpPost]
[Consumes(MediaTypeNames.Application.Json)]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public async Task < IActionResult > CreateAsync(Product product) {
  if (product.Description.Contains("XYZ Widget")) {
    return BadRequest();
  }

  await _repository.AddProductAsync(product);

  return CreatedAtAction(nameof(GetById), new {
    id = product.Id
  }, product);
}
```

API documentation with Swagger / OpenAPI

What is Swagger (OpenAPI)?

- Swagger (OpenAPI) is a language-agnostic specification for describing REST APIs.
- It allows both computers and humans to understand the capabilities of a REST API without direct access to the source code.
- Its main goals are to:
  - Minimize the amount of work needed to connect decoupled services.
  - Reduce the amount of time needed to accurately document a service.
  - OpenAPI is a specification.
- Swagger is tooling that uses the OpenAPI specification. For example, OpenAPIGenerator and SwaggerUI.

Swagger UI

- Swagger UI offers a web-based UI that provides information about the service, using the generated OpenAPI specification.

Swashbuckle Installation

- From the Manage NuGet Packages dialog:
  - Right-click the project in Solution Explorer > Manage NuGet Packages
  - Set the Package source to "nuget.org"
  - Ensure the "Include prerelease" option is enabled
  - Enter "Swashbuckle.AspNetCore" in the search box
  - Select the latest "Swashbuckle.AspNetCore" package from the Browse tab and click Install

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<TodoContext>(opt => opt.UseInMemoryDatabase("TodoList"));
    services.AddControllers();

    // Register the Swagger generator, defining 1 or more Swagger documents
    services.AddSwaggerGen();
}
```

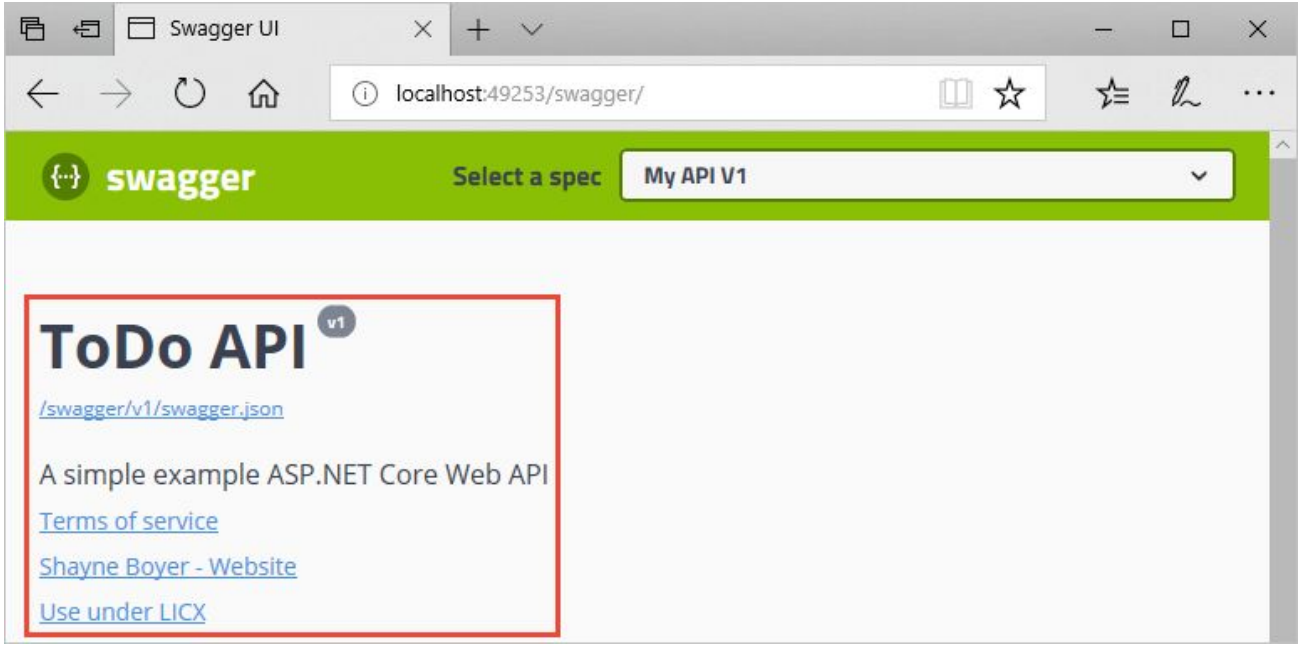
Swashbuckle Setup

```
public void Configure(IApplicationBuilder app)
{
    // Enable middleware to serve generated Swagger as a JSON endpoint.
    app.UseSwagger();

    // Enable middleware to serve swagger-ui (HTML, JS, CSS, etc.),
    // specifying the Swagger JSON endpoint.
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });

    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

API info and description (1 of 2)

```
using Microsoft.OpenApi.Models;

services.AddSwaggerGen(c =>
{
    c.SwaggerDoc("v1", new OpenApiInfo
    {
        Version = "v1",
        Title = "ToDo API",
        Description = "A simple example ASP.NET Core Web API",
        TermsOfService = new Uri("https://example.com/terms"),
        Contact = new OpenApiContact
        {
            Name = "Shayne Boyer",
            Email = string.Empty,
            Url = new Uri("https://twitter.com/spboyer"),
        },
        License = new OpenApiLicense
        {
            Name = "Use under LICX",
            Url = new Uri("https://example.com/license"),
        }
    });
});
```

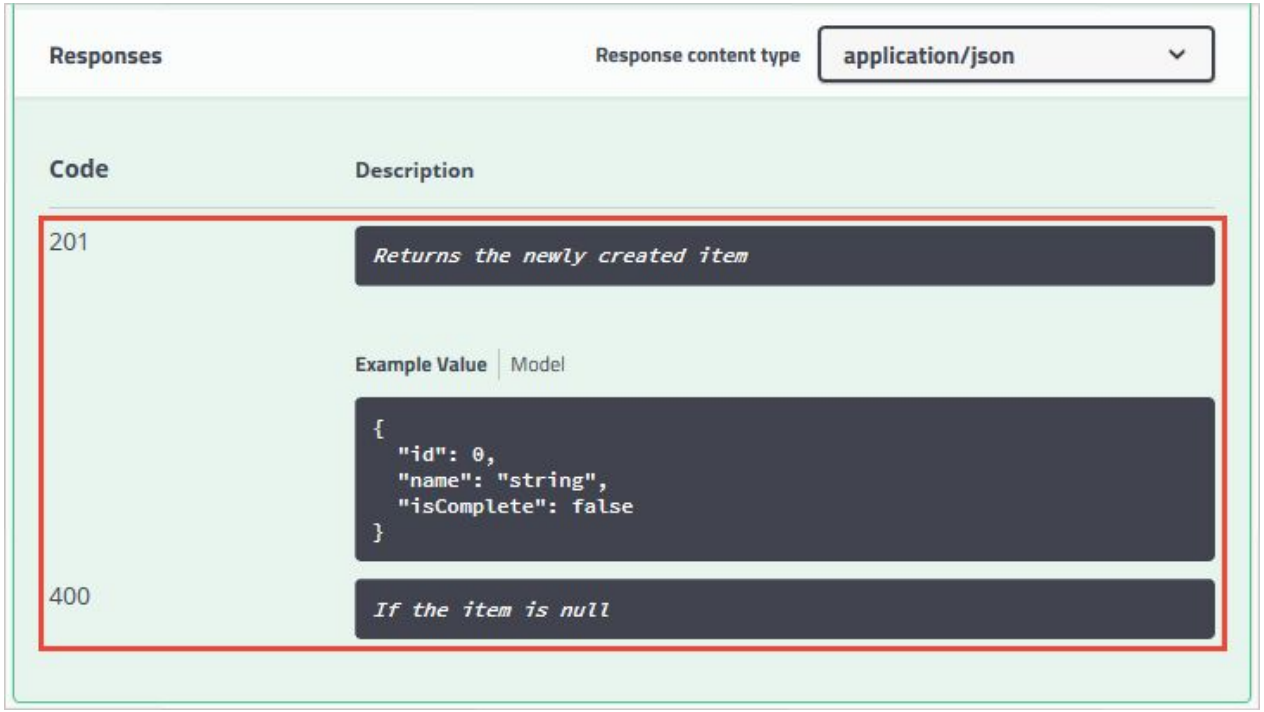
API info and description (2 of 2)



Describe response types (1 of 2)

```
/// <summary>
/// Creates a TodoItem.
/// </summary>
/// <remarks>
/// Sample request:
///
///     POST /Todo
///     {
///        "id": 1,
///        "name": "Item1",
///        "isComplete": true
///     }
///
/// </remarks>
/// <param name="item"></param>
/// <returns>A newly created TodoItem</returns>
/// <response code="201">Returns the newly created item</response>
/// <response code="400">If the item is null</response>
[HttpPost]
[ProducesResponseType(StatusCodes.Status201Created)]
[ProducesResponseType(StatusCodes.Status400BadRequest)]
public ActionResult<TodoItem> Create(TodoItem item)
{
    _context.TodoItems.Add(item);
    _context.SaveChanges();

    return CreatedAtRoute("GetTodo", new { id = item.Id }, item);
}
```

Describe response types (2 of 2)



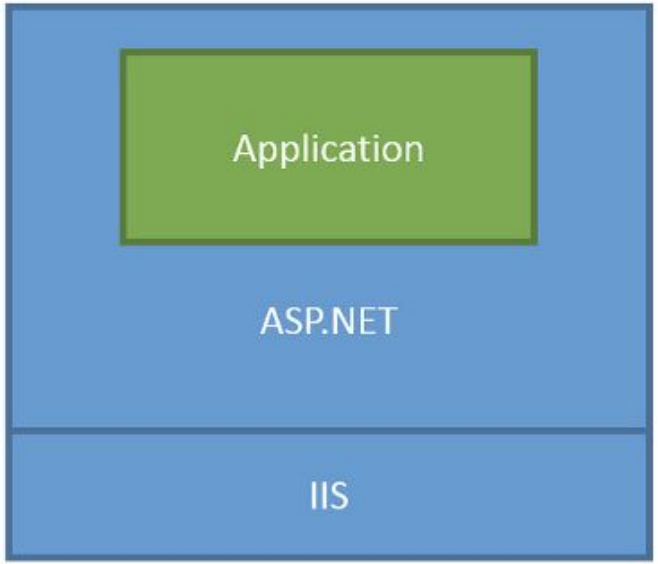
Middleware

What is Middleware?

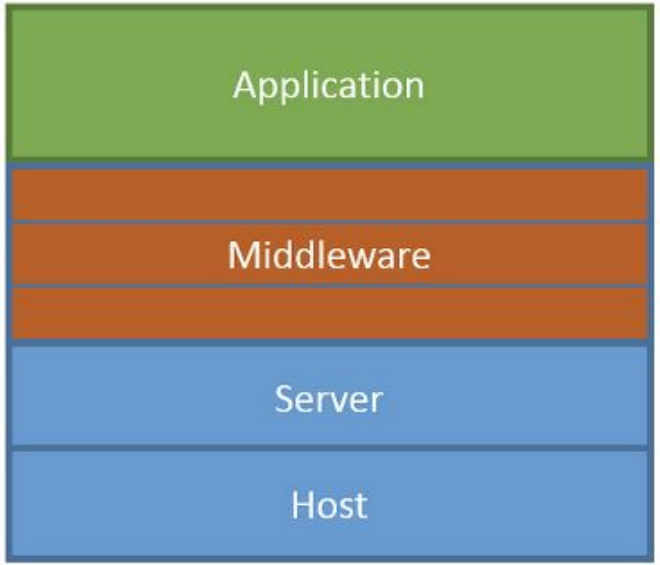
- Middleware is software that's assembled into an app pipeline to handle requests and responses
- Each component:
  - Chooses whether to pass the request to the next component in the pipeline.
  - Can perform work before and after the next component in the pipeline.

ASP.NET Core Middleware

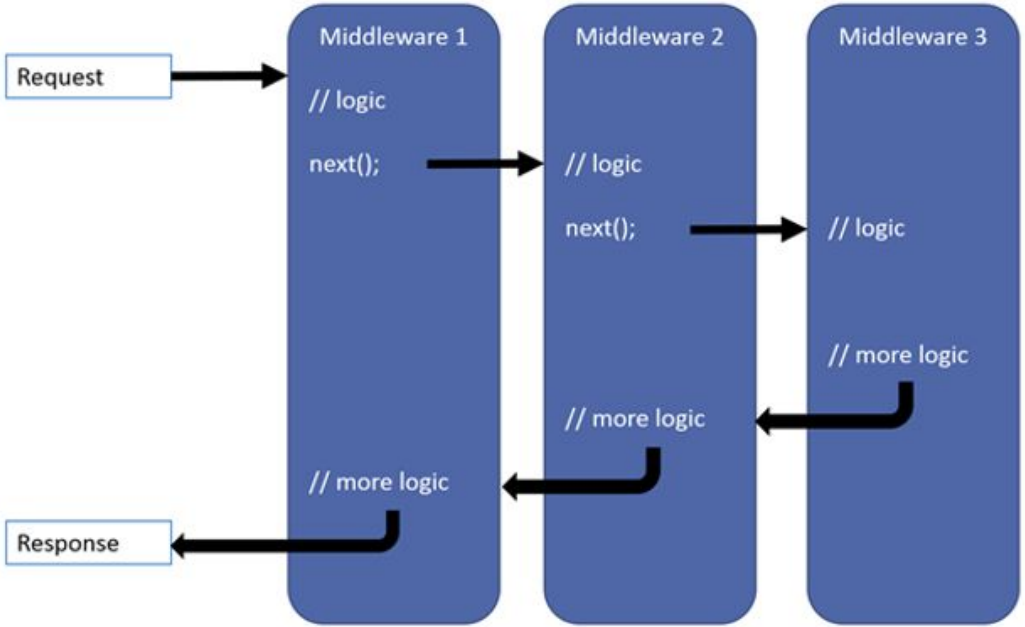
Traditional ASP.NET Application Model



ASP.NET Core Middleware



ASP.NET Core Middleware Execution



Basic Middleware

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello, World!");
        });
    }
}
```

Chaining Middlewares

- the Run delegate writes "Hello from 2nd delegate." to the response and then terminates the pipeline.
- If another Use or Run delegate is added after the Run delegate, it's not called.

```
public class Startup
{
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (context, next) =>
        {
            // Do work that doesn't write to the Response.
            await next.Invoke();
            // Do logging or other work that doesn't write to the Response.
        });

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from 2nd delegate.");
        });
    }
}
```

Branch the middleware pipeline (1 of 3)

- Map extensions are used as a convention for branching the pipeline.
- Map branches the request pipeline based on matches of the given request path.
- If the request path starts with the given path, the branch is executed.

Branch the middleware pipeline (2 of 3)

```
public class Startup
{
    private static void HandleMapTest1(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 1");
        });
    }

    private static void HandleMapTest2(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map Test 2");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1", HandleMapTest1);

        app.Map("/map2", HandleMapTest2);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

Branch the middleware pipeline (3 of 3)

- The following table shows the requests and responses from http://localhost:1234 using the previous code.

Request	Response
localhost:1234	Hello from non-Map delegate.
localhost:1234/map1	Map Test 1
localhost:1234/map2	Map Test 2
localhost:1234/map3	Hello from non-Map delegate.

Nesting Middleware

```
app.Map("/level1", level1App => {
    level1App.Map("/level2a", level2AApp => {
        // "/level1/level2a" processing
    });
    level1App.Map("/level2b", level2BApp => {
        // "/level1/level2b" processing
    });
});
```

Middleware Segment Support

```
public class Startup
{
    private static void HandleMultiSeg(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            await context.Response.WriteAsync("Map multiple segments.");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.Map("/map1/seg1", HandleMultiSeg);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate.");
        });
    }
}
```

Middleware MapWhen (1 of 2)

```
public class Startup
{
    private static void HandleBranch(IApplicationBuilder app)
    {
        app.Run(async context =>
        {
            var branchVer = context.Request.Query["branch"];
            await context.Response.WriteAsync($"Branch used = {branchVer}");
        });
    }

    public void Configure(IApplicationBuilder app)
    {
        app.MapWhen(context => context.Request.Query.ContainsKey("branch"),
                               HandleBranch);

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from non-Map delegate. <p>");
        });
    }
}
```

Middleware MapWhen (2 of 2)

- The following table shows the requests and responses from http://localhost:1234 using the previous code:

Request	Response
localhost:1234	Hello from non-Map delegate.
localhost:1234/?branch=main	Branch used = main

Middleware UseWhen (1 of 2)

- UseWhen also branches the request pipeline based on the result of the given predicate.
- Unlike with MapWhen, this branch is re-joined to the main pipeline if it doesn't short-circuit or contain a terminal middleware:

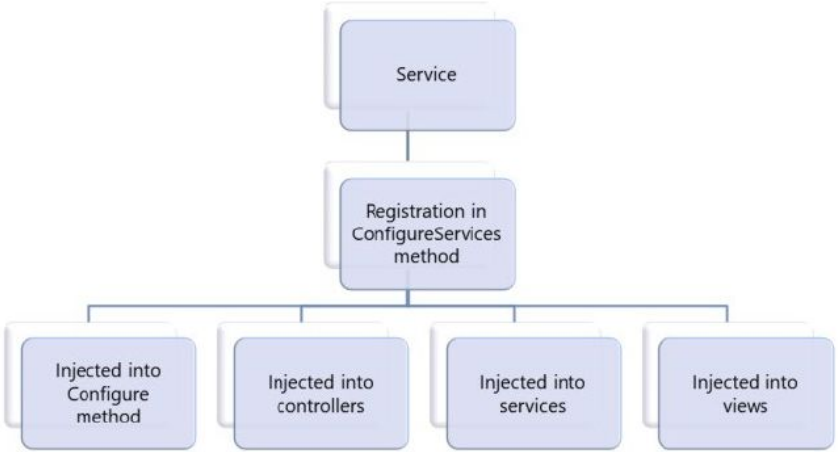
Middleware UseWhen (2 of 2)

```
public class Startup
{
    private void HandleBranchAndRejoin(IApplicationBuilder app, ILogger<Startup> logger)
    {
        app.Use(async (context, next) =>
        {
            var branchVer = context.Request.Query["branch"];
            logger.LogInformation("Branch used = {branchVer}", branchVer);

            // Do work that doesn't write to the Response.
            await next();
            // Do other work that doesn't write to the Response.
        });
    }

    public void Configure(IApplicationBuilder app, ILogger<Startup> logger)
    {
        app.UseWhen(context => context.Request.Query.ContainsKey("branch"),
                               appBuilder => HandleBranchAndRejoin(appBuilder, logger));

        app.Run(async context =>
        {
            await context.Response.WriteAsync("Hello from main pipeline.");
        });
    }
```

Configuring Services using Dependency Injection

Introduction to Dependency Injection

- Dependency Injection is a technique by which it is possible to facilitate separation of concerns in the application



Dependency Chain

```
public class MyClass\n
{\n
    private MyDependency _myDependency;\n
    public MyClass(MyDependency myDependency)\n
    {\n
        _myDependency = myDependency;\n
    }\n
}\n
public class MyDependency\n
{\n
    private MySubDependency _mySubDependency;\n
    public MyDependency(MySubDependency mySubDependency)\n
    {\n
        _mySubDependency = mySubDependency;\n
    }\n
}\n
public class MySubDependency\n
{\n
    public MySubDependency()\n
    {\n
    }\n
}\n

class Program\n
{\n
    static void Main(string[] args)\n
    {\n
        MySubDependency subDependency = new MySubDependency();\n
        MyDependency dependency = new MyDependency(subDependency);\n
        MyClass _myClass = new MyClass(dependency);\n
    }\n
}\n
```

Loosely coupled dependency chain

```
public interface IMyClass\n
{\n
}\n

public interface IMyDependency\n
{\n
}\n

public interface IMySubDependency\n
{\n
}\n

public class MyClass : IMyClass\n
{\n
    private IMyDependency _myDependency;\n
    public MyClass(IMyDependency myDependency)\n
    {\n
        _myDependency = myDependency;\n
    }\n
}\n

public class MyDependency : IMyDependency\n
{\n
    private IMySubDependency _mySubDependency;\n
    public MyDependency(IMySubDependency mySubDependency)\n
    {\n
        _mySubDependency = mySubDependency;\n
    }\n
}\n

public class MySubDependency : IMySubDependency\n
{\n
    public MySubDependency()\n
    {\n
    }\n
```

Using the Startup Class to Configure Services

- Any class can act as a service
- By using the **ConfigureServices** method, you can register any service you would like to use in the application
- By using Dependency Injection in the **Configure** method you can utilize the services inside the middleware

Injecting Custom Services

```
public interface IMyService
{
    string DoSomething();
}

public class MyService : IMyService
{
    public string DoSomething()
    {
        return "something";
    }
}
```

Inject Services to Controllers

- A controller is used to handle requests from the client
- Controllers support constructor dependency injection
- If the internal behavior of a service or its dependencies change, you will not need to update the controller
- You cannot have more then one constructor in the controller, as the default dependency injection container cannot handle it

Service Lifetime

- AddSingleton – Instantiates once in the application's lifetime
- AddScoped – Instantiates once per request made to the server
- AddTransient – Instantiates every single time the service is injected

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IFirstService, FirstService>();
    services.AddScoped<ISecondService, SecondService>();
    services.AddTransient<IThirdService, ThirdService>();
}
```

Routing

Configuring Routes by Using Attributes

- Attribute-based routing allows you to configure routes using attributes
- It allows you to define your routes in the same file as the controller that they refer to

```
[Route("Some")]
public IActionResult SomeMethod()
{
    return Content("Some method");
}
```

- Convention-based routing and attribute-based routing can be used in the same application

StartUp.Configure

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddControllers();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.UseHttpsRedirection();

    app.UseRouting();

    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Attribute-based routing

```
public class MyController : Controller
{
   [Route("Some")]
     public IActionResult SomeMethod()
     {
         return Content("Some method");
     }
}
```

An attribute route with parameter

```
public class MyController : Controller
{
    [Route("My/{param}")]
    public IActionResult SomeMethod(string param)
    {
        return Content(param);
    }
}
```

An attribute route with two parameters

```
public class MyController : Controller
{
    [Route("My/{param1}/{param2:int}")]
    public IActionResult SomeMethod(string param1, int param2)
    {
        return Content("param1: " + param1 + ", param2: " + param2);
    }
}
```

An attribute route with an optional parameter

```
public class MyController : Controller
{
    [Route("My/{param1}/{param2?}")]
    public IActionResult SomeMethod(string param1, string param2)
    {
        return Content("param1: " + param1 + ", param2: " + param2);
    }
}
```

Reserved routing names (1 of 2)

The following keywords are reserved route parameter names when using Controllers or Razor Pages:

- action
- area
- controller
- handler
- page

Reserved routing names (2 of 2)

- Using page as a route parameter with attribute routing is a common error.
- Doing that results in inconsistent and confusing behavior with URL generation.

```
public class HomeController : ControllerBase
{
    [Route("/articles/{page}")]
    [HttpGet]
    public ActionResult ListArticles(int page)
    {
        return StatusCode(200, $"You are trying to access page {page}");
    }
}
```

HTTP verb templates

ASP.NET Core has the following HTTP verb templates:

- [HttpGet]
- [HttpPost]
- [HttpPut]
- [HttpDelete]
- [HttpHead]
- [HttpPatch]

Attribute routing with HttpGet attributes

```
[Route("api/[controller]")]
[ApiController]
public class TestController : ControllerBase
{
    [HttpGet]    // GET /api/test
    public IActionResult ListProducts()
    {
        return StatusCode(200);
    }

    [HttpGet("{id}")]    // GET /api/test/xyz
    public IActionResult GetProduct(string id)
    {
        return StatusCode(200);
    }

    [HttpGet("int/{id:int}")] // GET /api/test/int/3
    public IActionResult GetIntProduct(int id)
    {
        return StatusCode(200);
    }

    [HttpGet("int2/{id}")]  // GET /api/test/int2/3
    public IActionResult GetInt2Product(int id)
    {
        return StatusCode(200);
    }
}
```

Two actions that match the same route template

```
[ApiController]\n
public class MyProductsController : ControllerBase\n
{\n
    [HttpGet("/products3")]\n
    public ActionResult ListProducts()\n
    {\n
        return StatusCode(200);\n
    }\n

    [HttpPost("/products3")]\n
    public ActionResult CreateProduct(Product product)\n
    {\n
        return StatusCode(201);\n
    }\n
}\n
```

Route name

Route names can be used to generate a URL based on a specific route. Route names:

- Have no impact on the URL matching behavior of routing.
- Are only used for URL generation.
- Route names must be unique application-wide.

```
[ApiController]\n
public class Products2ApiController : ControllerBase\n
{\n
    [HttpGet("/products2/{id}", Name = "Products_List")]\n
    public ActionResult GetProduct(int id)\n
    {\n
        return StatusCode(200);\n
    }\n
}\n
```

Combining Attribute Routes (1 of 2)

- Combine route attributes on the individual actions
- Any route templates defined on the controller are prepended to route templates on the actions
- Placing a route attribute on the controller makes all actions in the controller use attribute routing.

Combining Attribute Routes (2 of 2)

```
[ApiController]
[Route("products")]
public class ProductsApiController : ControllerBase
{
    [HttpGet]
    public ActionResult ListProducts()
    {
        return StatusCode(200);
    }

    [HttpGet("{id}")]
    public ActionResult GetProduct(int id)
    {
        return StatusCode(200);
    }
}
```

Non-existing Route

This localhost page can’t be found

No webpage was found for the web address: **https://localhost:5001/api/test5**

HTTP ERROR 404

Reload

Token replacement in route templates [controller] and [action] (1 of 3)

- Attribute routes support token replacement by enclosing a token in square-brackets ([,])
- The tokens [action] and [controller] are replaced with the values of the action name and controller name from the action where the route is defined

Token replacement in route templates [controller] and [action] (2 of 3)

```
[Route("[controller]/[action]")]
public class Products0Controller : Controller
{
    [HttpGet] // Matches /Products0/List
    public ActionResult List()
    {
        return StatusCode(200);
    }


    [HttpGet("{id}")] // Matches /Products0/Edit/{id}
    public ActionResult Edit(int id)
    {
        return StatusCode(200);
    }
}
```

Token replacement in route templates [controller] and [action] (3 of 3)

```
[ApiController]
[Route("api/[controller]/[action]", Name = "[controller]_[action]")]
public abstract class MyBase2Controller : ControllerBase
{
}

public class Products11Controller : MyBase2Controller
{
    [HttpGet]                       // /api/products11/
    public ActionResult List()
    {
        return StatusCode(200);
    }

    [HttpGet("{id}")]               // /api/products11/edit/3
    public ActionResult Edit(int id)
    {
        return StatusCode(200);
    }
}
```

Multiple attribute routes (1 of 2)

```
[Route("[controller]")]
public class Products13Controller : Controller
{
    [Route("")]      // Matches 'Products13'
    [Route("Index")] // Matches 'Products13/Index'
    public ActionResult Index()
    {
        return StatusCode(200);
    }
}
```

Multiple attribute routes (2 of 2)

```
[Route("Store")]
[Route("[controller]")]
public class Products6Controller : Controller
{
    [HttpPost("Buy")]        // Matches 'Products6/Buy' and 'Store/Buy'
    [HttpPost("Checkout")]   // Matches 'Products6/Checkout' and 'Store/Checkout'
    public ActionResult Buy()
    {
        return StatusCode(200);
    }
}
```

Handle errors in ASP.NET Core web APIs

Error

- In ASP.NET Core 3.0 and later, the Developer Exception Page displays a plain-text response if the client doesn't request HTML-formatted output. The following output appears:

```
HTTP/1.1 500 Internal Server Error
Transfer-Encoding: chunked
Content-Type: text/plain
Server: Microsoft-IIS/10.0
X-Powered-By: ASP.NET
Date: Fri, 27 Sep 2019 16:13:16 GMT

System.ArgumentException: We don't offer a weather forecast for chicago. (Parameter 'city')
   at WebApiSample.Controllers.WeatherForecastController.Get(String city) in C:\working_folde
   at lambda_method(Closure , Object , Object[] )
   at Microsoft.Extensions.Internal.ObjectMethodExecutor.Execute(Object target, Object[] para
   at Microsoft.AspNetCore.Mvc.Infrastructure.ActionMethodExecutor.SyncObjectResultExecutor.E
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.<InvokeActionMethodAsyn
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.<InvokeNextActionFilter
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Rethrow(ActionExecutedC
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.Next(State& next, Scope
   at Microsoft.AspNetCore.Mvc.Infrastructure.ControllerActionInvoker.InvokeInnerFilterAsync(
--- End of stack trace from previous location where exception was thrown ---
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeFilterPipelineAsync>g__A
   at Microsoft.AspNetCore.Mvc.Infrastructure.ResourceInvoker.<InvokeAsync>g__Logged|17_1(Res
   at Microsoft.AspNetCore.Routing.EndpointMiddleware.<Invoke>g__AwaitRequestTask|6_0(Endpoin
   at Microsoft.AspNetCore.Authorization.AuthorizationMiddleware.Invoke(HttpContext context)
   at Microsoft.AspNetCore.Diagnostics.DeveloperExceptionPageMiddleware.Invoke(HttpContext co

HEADERS
=======
Accept: */*
Host: localhost:44312
User-Agent: curl/7.55.1
```

Error Handler (1 of 2)

In `Startup.Configure`, invoke UseExceptionHandler to use the middleware:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }
    else
    {
        app.UseExceptionHandler("/error");
    }

    app.UseHttpsRedirection();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}
```

Error Handler (2 of 2)

Configure a controller action to respond to the `/error` route:

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error")]
    public IActionResult Error() => Problem();
}
```

Environment-specific Exception Handling Middleware(1 of 2)

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
    {
        app.UseExceptionHandler("/error-local-development");
    }
    else
    {
        app.UseExceptionHandler("/error");
    }
}
```

Environment-specific Exception Handling Middleware(2 of 2)

```
[ApiController]
public class ErrorController : ControllerBase
{
    [Route("/error-local-development")]
    public IActionResult ErrorLocalDevelopment([FromServices] IWebHostEnvironment webHostEnvironment)
    {
        if (webHostEnvironment.EnvironmentName != "Development")
        {
            throw new InvalidOperationException("This shouldn't be invoked in non-development environments.");
        }

        var context = HttpContext.Features.Get<IExceptionHandlerFeature>();

        return Problem(detail: context.Error.StackTrace, title: context.Error.Message);
    }

    [Route("/error")]
    public IActionResult Error() => Problem();
}
```