Week 10 - WebAPI

Created by Dakshil Gorasiya

CORS

Cross-Origin Resource Sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate any origins (domain, scheme, or port) other than its own from which a browser should permit loading resources.

CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request. In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request. (Used for PUT, PATCH, DELETE) (A request of type OPTION will be sent).

Working of cors in browser

Browser will set origin to current domain, server will response with header `Access-Control-Allow-Origin: allowed_origin,` then browser will match origin and server's response if they don't match browser will not send response to app.

In case of PUT, DELETE, PATCH request browser may set other headers like `Access-Control-Request-Method` , `Access-Control-Request-Headers` and server response with headers like `Access-Control-Allow-Origin` , `Access-Control-Allow-Methods` , `Access-Control-Allow-Headers` , `Access-Control-Max-Age`

https://learn.microsoft.com/en-us/aspnet/web-api/overview/security/enabling-cross-origin-requests-in-web-api

To enable CORS

install nuget pakage `Microsoft.AspNet.WebApi.Cors`

In App_start/WebApiConfig.cs write

```
config.EnableCors();
```

To enable cors controller level

```
using System.Net.Http;
using System.Web.Http;
using System.Web.Http.Cors;

namespace WebService.Controllers
{
    [EnableCors(origins: "http://mywebclient.azurewebsites.net", headers: "*",
methods: "*")]
    public class TestController : ApiController
    {
        // Controller methods not shown...
    }
}
```

To enable cors at action level

```
public class ItemsController : ApiController
{
    public HttpResponseMessage GetAll() { ... }

    [EnableCors(origins: "http://www.example.com", headers: "*", methods:
"*")]
    public HttpResponseMessage GetItem(int id) { ... }
    public HttpResponseMessage Post() { ... }
    public HttpResponseMessage PutItem(int id) { ... }
}
```

To enable cors globally

```
public static class WebApiConfig
{
    public static void Register(HttpConfiguration config)
    {
        var cors = new EnableCorsAttribute("www.example.com", "*", "*");
        config.EnableCors(cors);
        // ...
    }
}
```

If you set the attribute at more than one scope, the order of precedence is:

1. Action
2. Controller
3. Global

JWT

JWT (JSON Web Token) is a compact, URL-safe way to **securely transmit claims (data)** between a client and a server.
A JWT has **3 parts**, separated by dots (`.`)

1. Header

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

2. Payload

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "admin": true,
  "iat": 1516239022
}
```

3. Signature

```
HMACSHA256(
  base64(header) + "." + base64(payload),
  secretKey
)
```

How JWT Authentication Works

1. User logs in with credentials
2. Server verifies credentials
3. Server creates JWT and sends it to client
4. Client stores token (localStorage / cookie)
5. Client sends JWT in every request:

```
Authorization: Bearer <JWT>
```

6. Server:

- Verifies signature
- Checks expiration
- Reads claims
- Grants or denies access

Jwt is not encrypted so not send sensitive data in it

Jwt is stateless so server never store it so it cannot revoke it

To configure it

1. Install necessary packages

```
Microsoft.Owin
Microsoft.Owin.Security
Microsoft.Owin.Security.Jwt
Microsoft.Owin.Host.SystemWeb
System.IdentityModel.Tokens.Jwt
```

2. Create a startup class

```
[assembly: OwinStartup(typeof(AuthenticationAuthorizationWebAPI.Startup))]

namespace AuthenticationAuthorizationWebAPI
{
    public class Startup
    {
        public void Configuration(IAppBuilder app)
        {
            // For more information on how to configure your application, visit http
s://go.microsoft.com/fwlink/?LinkID=316888
            var issuer = ConfigurationManager.AppSettings["jwtIssuer"];
            var audience = ConfigurationManager.AppSettings["jwtAudience"];
            var secret = Encoding.UTF8.GetBytes(ConfigurationManager.AppSettin
gs["jwtSecret"]);

            app.UseJwtBearerAuthentication(new JwtBearerAuthenticationOptions
            {
                AuthenticationMode = AuthenticationMode.Active,
                TokenValidationParameters = new TokenValidationParameters()
                {
                    ValidateIssuer = true,
                    ValidateAudience = true,
                    ValidateIssuerSigningKey = true,
                    ValidIssuer = issuer,
                    ValidAudience = audience,
                    IssuerSigningKey = new SymmetricSecurityKey(secret)
                },
            });

        }
    }
}
```

3. To create jwt tokens in controller

```
private string CreateJwtToken(string username, string role)
        {
            var secret = ConfigurationManager.AppSettings["jwtSecret"];
            var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(secre
t));
            var credentials = new SigningCredentials(key, SecurityAlgorithms.Hma
cSha256);

            var claims = new[]
            {
                new Claim(ClaimTypes.Name, username),
                new Claim(ClaimTypes.Role, role),
            };

            var token = new JwtSecurityToken(
                issuer: ConfigurationManager.AppSettings["jwtIssuer"],
                audience: ConfigurationManager.AppSettings["jwtAudience"],
                claims: claims,
                expires: DateTime.Now.AddHours(1),
                signingCredentials: credentials
                );

            return new JwtSecurityTokenHandler().WriteToken(token);
        }
```

4. To check if jwt token is valid in action add attribute

```
[Authorize]
```

5. To check role based access

```
[Authorize(Roles = "Admin")]
```

To store configs in web.config

```
<appSettings>
        <add key="jwtSecret" value="secret_key" />
        <add key="jwtIssuer" value="issuer" />
        <add key="jwtAudience" value="audience" />
    </appSettings>
```

Exception Handling

Basic way

It will not send stack trace if normal exception is thrown it will send stack trace in response

```
throw new HttpResponseException(
                Request.CreateResponse(
                      HttpStatusCode.Forbidden,
                      new { message = "Access denied" }
                   )
            );
```

Quick response control but hard to manage for large apps

Exception filters

A class that catches **unhandled exceptions** globally or per controller.

```
public class GlobalExceptionFilter : ExceptionFilterAttribute
   {
      public override void OnException(HttpActionExecutedContext context)
      {
         if (context.Exception is InvalidValueException)
         {
            context.Response = context.Request.CreateErrorResponse(
                  HttpStatusCode.BadRequest,
                  context.Exception.Message
               );
         }
         else
         {
            context.Response = context.Request.CreateErrorResponse(
               HttpStatusCode.InternalServerError,
               "An unexpected error occured"
            );
         }
```

```
        }
    }
```

To enable it for action add attribute

```
[GlobalExceptionFilter]
```

To enable it globally add given line to WebApiConfig.cs

```
config.Filters.Add(new GlobalExceptionFilter());
```

It cannot handle pipeline error like routing error

Global Exception Handler

Catches **almost all exceptions**

```
public class GlobalExceptionHandler : ExceptionHandler
    {
        public override void Handle(ExceptionHandlerContext context)
        {
            context.Result = new ResponseMessageResult(
                    context.Request.CreateErrorResponse(
                            HttpStatusCode.InternalServerError,
                            "Exception handled globally"
                        )
                );
        }
    }
```

In WebApiConfig.cs

```
config.Services.Replace(
                typeof(IExceptionHandler),
            new GlobalExceptionHandler()
        );
```

Caching

Caching improves **performance, scalability, and response time** by storing frequently used data and avoiding repeated computation or database calls.

Client side caching

Client side caching is done by setting headers

Headers used

1. max-age=3600 → Cache for 1 hour
2. no-cache → caching is not allowed
3. no-store → caching not allowed (strictly)
4. public → can be cached by browser & proxies
5. private → Only browser can cache

Ex:

```
Cache-Control: public, max-age=3600
```

6. E-tag → By setting e-tag (Browser send it with every request in header If-None-MatchLas if it receive 304 Not modified it will use cached version otherwise store response)
7. Last-Modified: Timestamp (Browser send If-Modified-Since)

To set headers manually for request

```
public IHttpActionResult GetProducts()
{
    var response = Request.CreateResponse(HttpStatusCode.OK, products);

    response.Headers.CacheControl = new CacheControlHeaderValue
    {
        Public = true,
        MaxAge = TimeSpan.FromMinutes(10)
    };

    return ResponseMessage(response);
}
```

To create attribute to set headers

```
public class CacheHeaderAttribute : ActionFilterAttribute
{
    public int Duration { get; set; }

    public override void OnActionExecuted(HttpActionExecutedContext contex
t)
    {
        context.Response.Headers.CacheControl =
            new CacheControlHeaderValue
            {
                Public = true,
                MaxAge = TimeSpan.FromSeconds(Duration)
            };
    }
}
```

```
[CacheHeader(Duration = 300)]
```

To use e-tag

```
public HttpResponseMessage getEtagData()
      {
         var currentEtag = "\"prod-10-v2\"";

         if(Request.Headers.IfNoneMatch.Any(e ⇒ e.Tag == currentEtag))
         {
            return new HttpResponseMessage(HttpStatusCode.NotModified);
         }

         var response = Request.CreateResponse(HttpStatusCode.OK, "value");
         response.Headers.ETag = new System.Net.Http.Headers.EntityTagHea
derValue(currentEtag);
         response.Headers.CacheControl = new System.Net.Http.Headers.Cach
eControlHeaderValue
         {
            Private = true,
            NoCache = true
         };

         return response;
      }
```

In memory cache

Install package : System.Runtime.Caching

```
 public IEnumerable<string> Get()
      {
         ObjectCache cache = MemoryCache.Default;
```

```
            string cacheKey = "uniqueKey";

            string[] response = cache[cacheKey] as string[];

            if (response != null)
            {
                return new string[] { response[0], "Cache hit" };
            }

            // add if not exists, update else
            cache.Set(cacheKey, new string[] { "value1", "value2" }, DateTimeOffse
t.Now.AddMinutes(1));
             return new string[] { "value1", "value2" };
        }
```

Use MemoryCache.Default property to get reference of cache

Other methods of MemoryCache

Method	Description
Add	Adds an entry only if it does not already exist
AddOrGetExisting	Adds an entry if it does not exist; otherwise returns the existing value
Contain	Checks if a key exists
CreateCacheEntryChangeMonitor	Creates cache monitors; pass an array of keys—if they change (remove/add), the current cache entry is removed. Can be added via `policy.ChangeMonitors.Add`
Dispose	Frees resources
Get	Fetches an entry
GetCacheItem	Fetches the entire cache item
GetCount	Returns the total number of cache entries
GetEnumerator	Enumerates over key-value pairs

Method	Description
GetValues	Fetches multiple entries at a time
Remove	Removes an entry
Trim	Removes a given percentage of entries (Least Recently Used)

Redis

Redis (REmote DIctionary Server) is **a blazing-fast, open-source, in-memory data structure store used as a database, cache, message broker, and streaming engine**, known for its extreme speed (sub-millisecond latency) by keeping data in RAM, while also offering optional disk persistence and support for complex data types like strings, hashes, lists, and JSON, making it ideal for high-performance applications needing low latency.

To use redis

1. Install packages : StackExchange.Redis
2. Configure redis connection

```
using StackExchange.Redis;

public static class RedisConnectionHelper
{
    private static readonly Lazy<ConnectionMultiplexer> lazyConnection =
        new Lazy<ConnectionMultiplexer>(() ⇒
        {
            return ConnectionMultiplexer.Connect("localhost:6379");
        });

    public static ConnectionMultiplexer Connection ⇒ lazyConnection.Value;
}
```

3. Get redis instance

```
IDatabase redisDb = RedisConnectionHelper.Connection.GetDatabase();
```

4. Redis operation
   a. Set data

```
    redisDb.StringSet("user:1", "Dakshil", TimeSpan.FromMinutes(10));
```

   b. Get data

```
var value = redisDb.StringGet("user:1");
```

   c. Delete data

```
redisDb.KeyDelete("user:1");
```

Other Datatypes

1. String

Text, numbers, JSON, tokens, flags

Used for:

- Caching API responses
- JWT blacklist
- Feature flags

To set

```
redisDb.StringSet("string", "HELLO");
```

To get

```
redisDb.StringGet("string");
```

2. Hash

Best for storing object properties

Better than JSON when:

- want to update individual fields
- want less serialization overhead

To get entire object

```
HashEntry[] cachedData = redisDb.HashGetAll("hash");
```

To get a value

```
redisDb.HashGet("hash", "name");
```

To set hash

```
redisDb.HashSet("hash", new HashEntry[]
                {
                    new HashEntry("id", 1),
                    new HashEntry("name", "abc")
                });
```

3. List

Queue / Stack behavior

Used for:

- Background jobs
- Message queues

To leftpop, rightpop also available

```
RedisValue cachedData = redisDb.ListLeftPop("list");
```

To rightpush, leftpush also available

```
redisDb.ListRightPush("list", "1");
```

To get size

```
redisDb.LitLength();
```

4. Set

Unique unordered items

Used for:

- Tracking logged-in users
- Feature access lists

To check is set is contains a item

```
bool isExits = redisDb.SetContains("set", "1");
```

To remove from set

```
redisDb.SetRemove("set", "1");
```

To add item to set

```
redisDb.SetAdd("set", "1");
```

5. SortedSet

Ranking, leaderboards

Used for:

- Rate limiting
- Priority queues

To check if sortedset is not empty

```
long count = redisDb.SortedSetLength("sortedSet");
```

To get data between given range (inclusive)

```
var cachedData = redisDb.SortedSetRangeByRank("sortedSet", 1, 3);
```

To add item in sortedset

```
redisDb.SortedSetAdd("sortedSet", "a", 10);
```

API Versioning

API versioning is the practice of **managing changes in your API over time** so that existing clients don't break when you introduce new features, fix bugs, or change behaviors.

1 Url based versioning

GET /api/v1/products
GET /api/v2/products

To implement it use attribute based routing

```
[RoutePrefix("api/v1/data")]
   public class DataV1Controller : ApiController
   {
      [HttpGet]
      [Route("")]
      public IHttpActionResult Get()
      {
         return Ok("HI");
      }
   }
```

2 Query parameter based

GET /api/products?version=1
GET /api/products?version=2

To implement it

```
public IHttpActionResult Get()
      {
         var query = Request.GetQueryNameValuePairs();
         var versionParam = query.FirstOrDefault(q => q.Key == "api-version").
Value;

         if(versionParam == "1")
         {
            return Ok("V1");
         }
         else if(versionParam == "2")
         {
            return Ok("V2");
         }
         else
         {
            return Ok("Invalid version");
        }
    }
```

3 Header based versioning

GET /api/products
Header: API-Version: 1
Header: API-Version: 2

To implement it

```
public IHttpActionResult Get()
    {
        IEnumerable<string> versionHeaders;

        if (!Request.Headers.TryGetValues("X-API-Version", out versionHeader
s))
        {
            return BadRequest("API version header missing");
        }

        var version = versionHeaders.FirstOrDefault();

        if (version == "1")
        {
            return Ok("V1");
        }
        else if (version == "2")
        {
            return Ok("V2");
        }
        else
        {
            return Ok("Invalid API version");
        }
    }
```

Swagger

Swagger is used for API documentation and testing

Install package : Swashbuckle

To send bearer token configure SwaggerConfig.cs as given below

```
GlobalConfiguration.Configuration
            .EnableSwagger(c ⇒
             {
               c.ApiKey("Token")
                     .Description("Filling in the value: Bearer {your JWT token}")
                     .Name("Authorization")
                     .In("header");
             })
            .EnableSwaggerUi(c ⇒
             {
               c.EnableApiKeySupport("Authorization", "header");
             });
```

To get xml comments

Go to project property → Build → output → tick XML documentation file

```
c.IncludeXmlComments(GetXmlCommentsPath());
private static string GetXmlCommentsPath()
      {
         return string.Format(@"{0}\bin\AuthenticationAuthorizationWebAPI.xm
l", System.AppDomain.CurrentDomain.BaseDirectory);
        }
```

Postman

Postman is used for api testing

In postman we can create collection,folders to organize apis

We can create environments to store common variables

We can manually set headers, query params, cookies

Write script which run before and after api call

Logging

Logging is very useful to identify bug and resolve it

We store logs so it can be refer in future

There are 5 levels of logs

LogLevel	Severity	Typical Use
Trace	Most verbose level. Used for development and seldom enabled in production.	Ex. Request-payload, Response-payload, Begin-method-X or End-method-X
Debug	Debugging the application behavior from internal events of interest.	Ex. Executed query, User authenticated, Session expired
Info	Information that highlights progress or application lifetime events.	
Warn	Warnings about validation issues or temporary failures that can be recovered.	

LogLevel	Severity	Typical Use
Error	Errors where functionality has failed or Exception have been caught.	
Fatal	Most critical level. Application is about to abort.	

To log INFO and ERROR in different file

1. Install package NLog, NLog.Web
2. Create NLog.config file at root

```
<?xml version="1.0" encoding="utf-8" ?>
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     autoReload="true"
     throwConfigExceptions="true">

   <!-- Targets →
   <targets>

      <!-- INFO LOG →
      <target xsi:type="File"
            name="infoFile"
            fileName="${basedir}/Logs/Info/info-${shortdate}.log"
            layout="${longdate}|${level}|${logger}|${message}" />

      <!-- ERROR LOG →
      <target xsi:type="File"
            name="errorFile"
            fileName="${basedir}/Logs/Error/error-${shortdate}.log"
            layout="${longdate}|${level}|${logger}|${message}|${exception:for
mat=tostring}" />

   </targets>
```

```
    <!-- RULES -->
    <rules>

        <!-- Info & Warn -->
        <logger name="*" minlevel="Info" maxlevel="Warn" writeTo="infoFile" /
>

        <!-- Error & Fatal -->
        <logger name="*" minlevel="Error" writeTo="errorFile" />

    </rules>
</nlog>
```

3. In Global.asax.cs register nlog

```
LogManager.Setup().LoadConfigurationFromFile("Nlog.config");
```

4. To log some data manually

```
public class ValuesController : ApiController
    {
        private static readonly Logger logger = LogManager.GetCurrentClassLog
ger();

        // GET api/values
        public IEnumerable<string> Get()
        {
            logger.Info("Get values api called");
            return new string[] { "value1", "value2" };
        }
    }
```

5. To handle exception globally create a Filter and register it

```
using NLog;
using System.Web.Http.Filters;

public class GlobalExceptionFilter : ExceptionFilterAttribute
{
    private static readonly Logger logger = LogManager.GetCurrentClassLogge
r();

    public override void OnException(HttpActionExecutedContext context)
    {
        logger.Error(context.Exception, "Unhandled exception");
    }
}
```

```
config.Filters.Add(new GlobalExceptionFilter());
```

To create difference logger for a controller

1. Create a separate config file

```
<nlog xmlns="http://www.nlog-project.org/schemas/NLog.xsd"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

    <targets>
        <target xsi:type="File" name="controllerFile"
                fileName="Logs/ControllerLogs/${shortdate}.log"
                layout="${longdate} | ${level:uppercase=true} | ${message} ${exce
ption:format=toString}" />
    </targets>

    <rules>
        <logger name="ControllerLogger*" minlevel="Info" writeTo="controllerFil
```

```
e" />
    </rules>
</nlog>
```

2. Load that in controller make sure that name is same as config

```
public class ValuesController : ApiController
    {
        // Logger instance
        private static readonly Logger logger = LogManager.GetCurrentClassLog
ger();

        private readonly Logger _controllerLogger;

        public ValuesController()
        {
            var configPath = System.IO.Path.Combine(AppDomain.CurrentDomain.
BaseDirectory, "ValueLogging.config");
            var factory = new LogFactory(new XmlLoggingConfiguration(configPat
h));
            _controllerLogger = factory.GetLogger("ControllerLogger");
        }

        // GET api/values
        public IEnumerable<string> Get()
        {
            // Log an informational message
            _controllerLogger.Info("Get values api called");
            return new string[] { "value1", "value2" };
        }
    }
```

Dependency Injection

Instead of a class **creating is own dependencies**, they are **provided (injected)** from outside is known as dependency injection

1. To use DI in .net framework install packages

```
Unity
Unity.WebAPI
```

2. Create UnityConfig.cs

```
using DependencyInjectionDemo.Interfaces;
using DependencyInjectionDemo.Services;
using System.Web.Http;
using Unity;
using Unity.Lifetime;
using Unity.WebApi;

namespace DependencyInjectionDemo
{
    public static class UnityConfig
    {
        public static void RegisterComponents()
        {
            // A container is a logical unit that holds registrations of types
            var container = new UnityContainer();

            // transient ( default )
            container.RegisterType<IMessageService, MessageService>();
            //container.RegisterType<IMessageService, MessageService>(new Tra
nsientLifetimeManager());

            // Set the dependency resolver for Web API to use Unity
            GlobalConfiguration.Configuration.DependencyResolver = new UnityD
ependencyResolver(container);
        }
    }
}
```

3. Register UnityConfig in Global.asax.cs after WebApiConfig registration

```
UnityConfig.RegisterComponents();
```

4. To use it

```
public class ValuesController : ApiController
    {
        // Dependency injection via constructor
        private readonly IMessageService _messageService;

        public ValuesController(IMessageService messageService)
        {
            // Assign the injected service to a private field
            _messageService = messageService;
        }
    }
```

To add singleton dependency

```
container.RegisterSingleton<IMessageService, MessageService>();
container.RegisterType<IMessageService, MessageService>(new ContainerC
ontrolledLifetimeManager());
```

To add scoped dependency

```
container.RegisterType<IMessageService, MessageService>(new Hierarchica
lLifetimeManager());
```

```
    private readonly Logger _controllerLogger;

    public ValuesController()
    {
        var configPath = System.IO.Path.Combine(AppDomain.CurrentDomain.Ba
seDirectory, "ValueLogging.config");
        var factory = new LogFactory(new XmlLoggingConfiguration(configPat
h));
        _controllerLogger = factory.GetLogger("ControllerLogger");
    }

    // GET api/values
    public IEnumerable<string> Get()
    {
        // Log an informational message
        _controllerLogger.Info("Get values api called");
        return new string[] { "value1", "value2" };
    }
```

Filters

Filters are attributes that run **before or after controller actions** to handle **cross-cutting concerns** like:

- Authorization
- Validation

- Logging
- Exception handling

Filter Execution Order

```
Request
 ↓
Message Handlers
 ↓
Routing
 ↓
Authorization Filters
 ↓
Action Filters (OnActionExecuting)
 ↓
Controller Action
 ↓
Action Filters (OnActionExecuted)
 ↓
Exception Filters (only if exception)
 ↓
Response
```

Types of filter

1. Authorization filter

Interface: IAuthorizationFilter

```
public class JwtAuthAttribute: AuthorizationFilterAttribute
   {
       public override void OnAuthorization(HttpActionContext actionContext)
```

```
        {
           // ...
          }
      }
```

2. Action Filter

Interface: IActionFilter

```
public class LogActionFilter: ActionFilterAttribute
    {
        public override void OnActionExecuting(HttpActionContext actionContex
t)
        {
            System.Diagnostics.Debug.WriteLine($"Action {actionContext.ActionD
escriptor.ActionName} starting at {DateTime.Now}");
        }

        public override void OnActionExecuted(HttpActionExecutedContext actio
nExecutedContext)
        {
            System.Diagnostics.Debug.WriteLine($"Action {actionExecutedContex
t.ActionContext.ActionDescriptor.ActionName} finished at {DateTime.Now}");
        }
    }
```

3. Exception Filter

Interface: IExceptionFilter

```
public class GlobalExceptionFilter: ExceptionFilterAttribute
    {
        public override void OnException(HttpActionExecutedContext context)
        {
            System.Diagnostics.Debug.WriteLine(context.Exception);

            context.Response = context.Request.CreateResponse(
                HttpStatusCode.InternalServerError,
                new
                {
                    Message = "Something went wrong on the server.",
                    Error = context.Exception.Message
                });
        }
    }
```

To use filter at action or controller write attribute name in []

To use filter for all request in WebApiConfig.cs

```
config.Filters.Add(new GlobalExceptionFilter());
```

We can also create filters by directly inheriting interfaces