**Web Development**

**Introduction to Web Development**

Web development involves building applications that run on web browsers, interacting with users through the internet. It consists of both front-end (UI/UX) and back-end (server-side) development. In the case of Web APIs, it's specifically about building backend services that provide endpoints for clients to communicate with (such as a mobile app, web browser, or another API).

In .NET Framework 4.8, Web API is used to build RESTful services. These services expose endpoints over HTTP, allowing clients to interact with resources (such as data or functionalities) over the web.

**Web API project**

A Web API project in .NET Framework 4.8 is typically created using the ASP.NET Web API template. This template provides the necessary setup to build a RESTful API with standard HTTP methods like GET, POST, PUT, DELETE, etc.

* Web API is used to expose your server-side logic to clients (like browsers or mobile apps) via HTTP requests.
* You define controllers and actions within those controllers to handle the requests.

The key structure typically includes:

* Controllers: Where the logic to handle requests resides.
* Models: Represent the data structure.
* Routing: Defines how HTTP requests map to controller actions.

**Building Web API**

1. **Start a New Project**:
   * Open Visual Studio.
   * Select **File > New > Project**.
   * Choose **ASP.NET Web Application (.NET Framework)**.
   * Name the project and click **OK**.
2. **Select Project Template**:
   * In the **New ASP.NET Project** dialog, choose **Empty**.
   * Enable **Web API** to include required libraries and configurations.
3. **Structure of Web API Project**:
   * **Controllers Folder**: Contains Web API controllers. Each controller handles HTTP requests.
   * **Models Folder**: Contains model classes representing the data structure.
   * **App\_Start Folder**: Contains configuration files like WebApiConfig.cs for routing.
   * **Web.config**: Configuration file for the application.
4. **Configuring Routes**:
   * Open WebApiConfig.cs (in the App\_Start folder).
   * Define the API routing using the MapHttpRoute method
5. **Creating a Controller:**
   * Add a new controller by right-clicking the Controllers folder and selecting Add > Controller.
   * Choose Web API Controller - Empty and name it (e.g., ProductsController).
6. **Adding Actions:**
   * Implement methods in the controller to handle HTTP requests
7. **Testing the API:**
   * Run the application and use a tool like Postman or Swagger to test your API endpoints.

**Action Method Response (HTTP status code etc.)**

Action methods in Web API return HTTP responses. The status code in the HTTP response indicates the outcome of the request.

**Common HTTP Status Codes:**

* **200 Ok()** : The request was successful (used with GET, PUT, POST, etc.)
* **201 Created()** : A resource was successfully created (used with POST)
* **302 Redirect()** : The resource has temporarily moved to a different URI (used with GET)
* **400 BadRequest()** : The request is invalid (e.g., missing parameters, malformed syntax)
* **401 Unauthorized()** : Authentication is required and has failed or not been provided
* **404 NotFound()** : The requested resource could not be found on the server
* **409 Conflict()** : There is a conflict with the current state of the resource (e.g., trying to create a resource that already exists)
* **500 InternalServerError()** : Something went wrong on the server, causing it to fail processing the request

**Security (CORS, Authentication, Authorization,Exception, JWT token etc.)**

Security is crucial when developing Web APIs. In .NET Framework, you can secure your API in several ways:

**CORS (Cross-Origin Resource Sharing)**

CORS allows a web application running on one domain to access resources on another domain. By default, browsers block cross-origin requests for security.

**Enabling CORS in Web API:**

1. Install the CORS package via NuGet:

Package : Microsoft.AspNet.WebApi.Cors

1. Enable CORS in WebApiConfig.cs:

config.EnableCors();

1. Apply CORS settings at the controller or action level:

Ex : [EnableCors(origins: "https://example.com", headers: "\*", methods: "GET,POST")]

**Basic Authentication in Web API**

**Basic Authentication** is a simple mechanism to verify the identity of users accessing your API. It involves sending a username and password with each HTTP request, encoded as a Base64 string.

**How It Works:**

1. The client sends an HTTP request with the Authorization header containing a Base64-encoded username and password.
2. The server decodes and validates the credentials.
3. If valid, the server grants access; otherwise, it rejects the request with a 401 Unauthorized status code.

**Steps to Implement Basic Authentication in Web API:**

1. **Create a Custom Authentication Handler:** Create a class that inherits from AuthorizeAttribute to handle authentication logic.

public class BasicAuthenticationAttribute : AuthorizationFilterAttribute

{

public override void OnAuthorization(HttpActionContext actionContext)

{

// logic

}

}

1. **Apply the Custom Attribute:** Use the BasicAuthenticationAttribute on controllers or specific action methods.

[BasicAuthentication]

public class ProductsController : ApiController

{

public IHttpActionResult Get()

{

return Ok(new { Message = "Welcome, authenticated user!" });

}

}

**Limitations of Basic Authentication:**

* Credentials are sent with every request, increasing the risk of exposure if not using HTTPS.
* Base64 encoding is not encryption, so data is easily decoded.
* Not suitable for sensitive applications unless combined with HTTPS.

**When to Use Basic Authentication:**

* For simple APIs with limited users.
* During development or for internal tools where security requirements are minimal.

**JWT Token Authentication**

**What is JWT?**

JWT (JSON Web Token) is a secure, compact, and self-contained way to transmit information between parties as a JSON object. It is commonly used in web APIs to handle authentication and secure user sessions in a stateless manner.

**How JWT Works**

1. **Client Login**:
   * The client sends their credentials (e.g., username and password) to the server.
2. **Token Generation**:
   * The server validates the credentials and generates a JWT containing user-specific data (e.g., user ID, roles).
3. **Token Storage**:
   * The client stores the token (e.g., in localStorage, sessionStorage, or cookies).
4. **Authorized Requests**:
   * For each request, the client sends the JWT in the Authorization header:

Ex : Authorization: Bearer <token>

1. **Server Validation**:
   * The server validates the token and grants/denies access based on its claims.

Implementing JWT in Web API Require :

* Package : System.IdentityModel.Tokens.Jwt
* Method to generate token
* Method to validate token
* Secure endpoints with custom or built in attribute

**Advantages of JWT**

1. **Stateless**: Tokens are self-contained; no need to store session data on the server.
2. **Compact**: Suitable for transmitting over HTTP headers.
3. **Interoperable**: Works across different platforms and programming languages.

**Best Practices**

1. **Use HTTPS**: Always transmit tokens over HTTPS to prevent interception.
2. **Short Expiry**: Set a short token expiration time and refresh it periodically.
3. **Strong Secret**: Use a strong, randomly generated secret key.
4. **Avoid Sensitive Data**: Do not store sensitive information in the token payload.

**HTTP caching**

Caching improves performance by storing copies of responses and serving them without contacting the server repeatedly. ASP.NET Web API supports output caching and client-side caching.

Types of Caching

1. Server-Side Caching: Store responses on the server.
2. Client-Side Caching: Use cache-related headers like Cache-Control.

Example of caching using Client-Side Caching with HttpContext.Cache :

**1. Adding Data to the Cache**

Use the Insert method to store data in the cache.

HttpContext.Current.Cache.Insert(key, value, null, DateTime.MaxValue, duration);

**2. Retrieving Data from the Cache**

Check if the data is already cached and retrieve it.

HttpContext.Current.Cache[key];

**3. Removing Data from the Cache**

To remove an item from the cache, use the Remove method.

HttpContext.Current.Cache.Remove(key);

**Key Parameters of Cache.Insert**

* **Key**: A unique identifier for the cached item.
* **Value**: The data to be cached.
* **Dependencies**: Optional dependencies, such as file paths or database changes.
* **Absolute Expiration**: A fixed point in time when the cache expires.
* **Sliding Expiration**: The cache duration resets with each access.

**Versioning**

Versioning in ASP.NET Framework 4.8 Web API is a practice that helps manage different versions of your API so that clients using older versions of the API can still function while newer versions with improved or changed functionality can be developed and exposed to clients. This ensures backward compatibility and allows for the introduction of new features without breaking existing client applications.

1. **URI Path Versioning**

This is the most common method where the version number is included in the URL path of the API.

Example:

* http://example.com/api/v1/products (Version 1)
* http://example.com/api/v2/products (Version 2)

Using 2 different controllers

Ex:   
// Version 1 Controller

public class ProductsV1Controller : ApiController

{

public IHttpActionResult Get()

{

return Ok("Version 1 Products");

}

}

// Version 2 Controller

public class ProductsV2Controller : ApiController

{

public IHttpActionResult Get()

{

return Ok("Version 2 Products");

}

}

**2. Query String Versioning**

In this approach, the version is passed in the query string.

**Example**:

* Version 1 URL: http://example.com/api/products?version=1
* Version 2 URL: <http://example.com/api/products?version=2>

Ex:

public class ProductsController : ApiController

{

public IHttpActionResult Get()

{

var version = Request.GetQueryNameValuePairs()

.FirstOrDefault(q => q.Key == "version").Value;

if (version == "1")

{

return Ok("Version 1 Products");

}

else if (version == "2")

{

return Ok("Version 2 Products");

}

return BadRequest("Invalid version");

}

}

**3. Header Versioning**

This method sends the version number in the request headers, which are typically hidden from the URL.

**Example**:

* Header: X-API-Version: 1
* Header: X-API-Version: 2

Ex:

public class ProductsController : ApiController

{

public IHttpActionResult Get()

{

var version = Request.Headers.GetValues("X-API-Version").FirstOrDefault();

if (version == "1")

{

return Ok("Version 1 Products");

}

else if (version == "2")

{

return Ok("Version 2 Products");

}

return BadRequest("Invalid version");

}

}

**Use of Swagger**

Swagger (now known as OpenAPI) is a framework for documenting and testing RESTful APIs. In .NET, you can use Swashbuckle to automatically generate Swagger documentation for your API.

* It creates an interactive UI where developers can explore your API's endpoints and make test calls.

Package : Swashbuckle

**Use of POSTMAN**

Postman is a tool used for testing Web APIs. With Postman, you can:

* Send HTTP requests to your API (GET, POST, PUT, DELETE).
* View the response status, headers, and body.
* Automate tests for APIs.

It's a great way to ensure that your API works as expected during development and debugging.

**Deployment**

Deployment involves taking your Web API and making it accessible to the public (or a private network). In the case of .NET Framework 4.8, typical deployment strategies include:

* IIS (Internet Information Services): Host your API on a Windows server with IIS.

Steps in Deployment:

1. Publish your Web API project from Visual Studio.
2. Configure your hosting environment (e.g., IIS, cloud service).
3. Ensure all dependencies (like databases) are correctly configured.