# .NET Web API Project Documentation

## Action Method Response

- \*\*HTTP Status Codes\*\*:   
 - Action methods in Web API return standard HTTP status codes to indicate the success or failure of an operation.  
 - Common status codes include:  
 - 200 OK: Successful request with data returned.  
 - 201 Created: Resource created successfully.  
 - 204 No Content: Successful request, no content returned.  
 - 400 Bad Request: Client-side input error.  
 - 401 Unauthorized: Authentication failed or not provided.  
 - 403 Forbidden: Access denied, even if authenticated.  
 - 404 Not Found: Resource not found.  
 - 500 Internal Server Error: Server-side error.  
- \*\*Best Practices\*\*:   
 - Use IHttpActionResult or HttpResponseMessage to standardize responses.  
 - Include meaningful error messages in response payloads for better debugging.  
- Example:

public IHttpActionResult Get(int id)  
{  
 if (id <= 0)  
 return BadRequest("Invalid ID.");  
  
 var item = repository.GetItem(id);  
 if (item == null)  
 return NotFound();  
  
 return Ok(item);  
}

## Security

- \*\*CORS (Cross-Origin Resource Sharing)\*\*:   
 - Allows or restricts API access from specific domains.  
 - Enabled using `EnableCors` attribute in Web API.  
 - Example:

[EnableCors(origins: "\*", headers: "\*", methods: "\*")]  
public class ValuesController : ApiController { }

- \*\*Authentication and Authorization\*\*:   
 - Use token-based authentication (e.g., JWT) for secure API access.   
 - Apply `[Authorize]` attribute for authorization.   
 - Example:

[Authorize]  
public IHttpActionResult GetSecureData()  
{  
 return Ok("This is secure data.");  
}

- \*\*Exception Handling\*\*:   
 - Use global exception handling with `ExceptionFilterAttribute` to manage errors.   
 - Example:

public class CustomExceptionFilter : ExceptionFilterAttribute  
{  
 public override void OnException(HttpActionExecutedContext context)  
 {  
 context.Response = new HttpResponseMessage(HttpStatusCode.InternalServerError)  
 {  
 Content = new StringContent("An error occurred."),  
 ReasonPhrase = "Critical Exception"  
 };  
 }  
}

## HTTP Caching

- Use caching headers to improve performance and reduce server load.  
- Example with `OutputCache` attribute:

[OutputCache(Duration = 60)]  
public IHttpActionResult GetData()  
{  
 return Ok("Cached data for 60 seconds.");  
}

## Versioning

- API versioning is essential to maintain backward compatibility.  
- Use `Namespace` or `RoutePrefix` to specify versions.  
- Example:

[RoutePrefix("api/v1/values")]  
public class ValuesV1Controller : ApiController { }

## Use of Swagger

- Swagger provides API documentation and testing interface.  
- Integrate Swagger using `Swashbuckle` NuGet package.  
- Example configuration in `Startup.cs`:

public class SwaggerConfig  
{  
 public static void Register(HttpConfiguration config)  
 {  
 config.EnableSwagger(c => c.SingleApiVersion("v1", "My API"))  
 .EnableSwaggerUi();  
 }  
}

## Use of POSTMAN

- POSTMAN is used to test APIs by sending requests and analyzing responses.  
- Steps:  
 1. Install POSTMAN.  
 2. Enter the API endpoint.  
 3. Select the HTTP method (GET, POST, etc.).  
 4. Add headers or body data as needed.  
 5. Send the request and review the response.

## Deployment

- Web API can be deployed to IIS or cloud platforms like Azure.  
- Steps for IIS Deployment:  
 1. Build the project in Release mode.  
 2. Publish the project from Visual Studio.  
 3. Configure IIS to host the published files.  
 4. Bind the domain or port.  
- For Azure:  
 1. Use Azure App Service.  
 2. Deploy using Visual Studio or Azure DevOps.