Api Basic Training

# Web Development

Prince Goswami

# 1. Introduction to Web Development

Web development is a dynamic field that encompasses the design, creation, and maintenance of websites and web applications. In today's digital age, a strong online presence is essential for businesses, organizations, and individuals, making web development a critical aspect of modern technology. This section provides a comprehensive overview of the fundamental concepts and elements within web development.

## 1.1 Key Components of Web Development

### 1.1.1 Front-End Development

Front-end development involves crafting the user interface and user experience of a website or web application. Developers use languages such as HTML, CSS, and JavaScript to create visually appealing and interactive interfaces that users can engage with directly.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Sample Website</title>
  <link rel="stylesheet" href="styles.css">
</head>
<body>
  <h1>Hello, World!</h1>
  <p>This is a sample webpage.</p>
  <script src="script.js"></script>
</body>
</html>
```

### 1.1.2 Back-End Development

Back-end development involves server-side logic and database management. It is responsible for handling requests from the front end, processing data, and ensuring the overall functionality of the web application.

### 1.1.3 Full-Stack Development

Full-stack developers are proficient in both front-end and back-end development. They possess a holistic understanding of web development, allowing them to work on all aspects of a project.

## 1.2 Technologies and Frameworks

### 1.2.1 Front-End Technologies

- HTML (HyperText Markup Language): Defines the structure of web content.
- CSS (Cascading Style Sheets): Styles and formats HTML elements.
- JavaScript: Adds interactivity and dynamic behavior to web pages.

### 1.2.2 Back-End Technologies

- Node.js: A JavaScript runtime for server-side development.
- ASP.NET: A framework for building scalable and robust web applications (used in .NET Framework 4.8).

## 1.3 Importance of Web Development

Web development plays a pivotal role in the digital landscape for several reasons:

- Global Reach: Websites and applications provide a global platform for businesses and individuals to reach a diverse audience.
- User Interaction: Interactive and responsive interfaces enhance user experience, leading to increased engagement.
- E-Commerce: Web development facilitates online transactions, contributing to the growth of e-commerce.
- Innovation: Continuous advancements in web technologies drive innovation, enabling the creation of dynamic and feature-rich applications.

## 1.4 Evolving Trends in Web Development

Web development is a dynamic field, constantly evolving to meet changing demands and embrace emerging technologies. Current trends include:

- Progressive Web Apps (PWAs): Combining the best of web and mobile apps for enhanced user experience.
- Single Page Applications (SPAs): Providing seamless, fluid user experiences by loading content dynamically.
- Serverless Architecture: Utilizing cloud services for scalable and cost-effective application deployment.

# 2. Web API Project in .NET Framework 4.8

In this section, we will dive into the specifics of creating a Web API project using the .NET Framework 4.8. A Web API project enables the development of scalable and interconnected applications by facilitating communication between different software systems over the web.

# 3 Building a Web API

## 3.1 Project Setup

To initiate a Web API project in .NET Framework 4.8, follow these steps:

Open Visual Studio:

Launch Visual Studio on your development machine.
Create a New Project:

Select "File" > "New" > "Project" from the menu.
Choose "ASP.NET Web Application" as the project template.
Select Web API Template:

In the project template selection window, choose "Web API" as the project template.
Click "Create" to initialize the project.

## 3.2 Controller and Routes

A Web API project revolves around controllers that handle incoming HTTP requests and produce appropriate responses.

Example: Sample Controller

```
public class ProductsController : ApiController
{
    // GET api/products
    public IEnumerable<Product> Get()
    {
        // Retrieve and return a list of products
    }

    // POST api/products
    public IHttpActionResult Post([FromBody] Product product)
    {
        // Create a new product
        // Return IHttpActionResult indicating success or failure
    }

    // DELETE api/products/1
    public IHttpActionResult Delete(int id)
    {
        // Delete a product by ID
        // Return IHttpActionResult indicating success or failure
    }
```

```
}
```

### 3.3 Model and Data Access

Define models to represent data entities and employ data access methods to interact with databases or other data sources.

Example: Product Model

```
public class Product
{
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Price { get; set; }
}
```

### 3.4 Action Method Response (HTTP Status Code, etc.)

HTTP status codes are crucial for conveying the result of a request. Appropriate use of status codes facilitates effective communication between clients and the Web API

Example: Action Method Responses

```
// GET api/products
public IHttpActionResult Get()
{
    var products = productService.GetAllProducts();
    if (products.Any())
    {
        return Ok(products); // 200 OK
    }
    else
    {
        return NotFound(); // 404 Not Found
    }
}
// POST api/products
public IHttpActionResult Post([FromBody] Product product)
{
    if (productService.AddProduct(product))
    {
        return Created("api/products", product); // 201 Created
    }
    else
    {
```

```
        return BadRequest("Invalid product data"); // 400 Bad Request
    }}
```

## 2.2 Action Method Response (HTTP Status Code, etc.) in .NET Framework 4.8 Web API

Ensuring proper handling of HTTP status codes is crucial in a Web API to convey the success or failure of a request. This section outlines examples of action method responses, including HTTP status codes and their meanings, in a .NET Framework 4.8 Web API project.

### 2.2.1 HTTP Status Codes Overview

HTTP status codes are three-digit numbers returned by the server to indicate the result of a client's request. Understanding and using these codes appropriately is essential for effective communication between the Web API and its clients.

Common HTTP Status Codes:

| Status Code | Description |
|---|---|
| 200 OK | The request was successful. |
| 201 Created | The request resulted in the creation of a new resource. |
| 204 No Content | The server successfully processed the request but is not returning any content. |
| 400 Bad Request | The request cannot be fulfilled due to bad syntax, invalid request message framing, or deceptive request routing. |
| 401 Unauthorized | The request lacks valid authentication credentials for the target resource. |
| 403 Forbidden | The server understood the request but refuses to authorize it. |
| 404 Not Found | The requested resource could not be found on the server. |
| 500 Internal Server Error | A generic error message returned when an unexpected condition was encountered by the server. |

### 2.2.2 Examples of Action Method Responses

**Retrieving a List of Products**

```
// GET api/products
public IHttpActionResult Get()
{
    var products = productService.GetAllProducts();
    if (products.Any())
    {
        return Ok(products); // 200 OK
    }
    else
    {
        return NotFound(); // 404 Not Found
    }
}
```

**Creating a New Product**

```
// POST api/products
public IHttpActionResult Post([FromBody] Product product)
{
    if (productService.AddProduct(product))
    {
        return Created("api/products", product); // 201 Created
    }
    else
    {
        return BadRequest("Invalid product data"); // 400 Bad Request
    }
}
```

**Updating an Existing Product**

```
// PUT api/products/1
public IHttpActionResult Put(int id, [FromBody] Product product)
{
    if (productService.UpdateProduct(id, product))
    {
        return Ok(product); // 200 OK
```

```
    }
    else
    {
        return NotFound(); // 404 Not Found
    }
}
```

**Deleting a Product**

```
// DELETE api/products/1
public IHttpActionResult Delete(int id)
{
    if (productService.DeleteProduct(id))
    {
        return StatusCode(HttpStatusCode.NoContent); // 204 No Content
    }
    else
    {
        return NotFound(); // 404 Not Found
    }
}
```

These examples demonstrate how to utilize common HTTP status codes in action methods to provide clear and meaningful responses to clients interacting with your .NET Framework 4.8 Web API. Proper status code usage enhances the reliability and effectiveness of your API.
.

# 4 Security in .NET Framework 4.8 Web API

Ensuring the security of your Web API is essential to protect data and resources. This section covers various security aspects, including CORS (Cross-Origin Resource Sharing), Authentication, Authorization, Exception handling, and the use of JWT (JSON Web Tokens) in a .NET Framework 4.8 Web API project.

## 4.1 CORS (Cross-Origin Resource Sharing)

CORS allows or restricts web applications running at one origin to make requests to a different origin. Configuring CORS is crucial for controlling cross-origin requests and enhancing the security of your Web API.

Example: CORS Configuration

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

In this example, only requests originating from "https://example.com" are allowed. Adjust the origins, methods, and headers based on your application's requirements.

## 4.2 Authentication and Authorization in .NET Framework 4.8 Web API

Authentication and authorization are crucial aspects of securing a Web API. .NET Framework 4.8 Web API provides flexibility in choosing authentication and authorization mechanisms. This section covers various types of authentication and authorization, including JWT (JSON Web Tokens), API keys, and role-based access control (RBAC).

### 4.2.1 JWT (JSON Web Tokens)

Example: JWT Token Generation

```
public class JwtTokenGenerator
{
    public static string GenerateToken(User user)
    {
        // ... (previous JWT example)
    }

    public static ClaimsPrincipal ValidateToken(string token)
    {
        var tokenHandler = new JwtSecurityTokenHandler();
        var validationParameters = GetValidationParameters();

        try
        {
            SecurityToken validatedToken;
            return tokenHandler.ValidateToken(token, validationParameters,
out validatedToken);
        }
        catch (Exception)
        {
            return null; // Token validation failed
        }
```

```
    }

    private static TokenValidationParameters GetValidationParameters()
    {
        return new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = "your-issuer",
            ValidAudience = "your-audience",
            IssuerSigningKey = new
SymmetricSecurityKey(Encoding.UTF8.GetBytes("YourSecretKeyHere"))
        };
    }
}
```

In this extended example, a method for validating JWT tokens is added. This method can be used in the authentication process.

### 4.2.2 API Key Authentication

API key authentication involves sending a unique key with each API request. The server verifies the key to authenticate the client.

Example: API Key Authentication

```
public class ApiKeyAuthenticationAttribute : Attribute,
IAuthenticationFilter
{
    public Task AuthenticateAsync(HttpAuthenticationContext context,
CancellationToken cancellationToken)
    {
        var apiKey =
context.Request.Headers.GetValues("X-API-Key").FirstOrDefault();

        if (IsValidApiKey(apiKey))
        {
            var identity = new GenericIdentity("APIKeyUser");
            context.Principal = new GenericPrincipal(identity, null);
        }
        else
        {
```

```
            context.ErrorResult = new UnauthorizedResult(new
AuthenticationHeaderValue[] { }, context.Request);
        }

        return Task.FromResult(0);
    }

    private bool IsValidApiKey(string apiKey)
    {
        // Implement your logic to validate the API key
        return apiKey == "YourValidAPIKey";
    }

    // Implement other IAuthenticationFilter methods...
}
```

This attribute can be applied to controllers or actions that require API key authentication.

#### 4.2.3 Role-Based Access Control (RBAC)

Role-based access control allows you to assign roles to users and control access based on these roles.

Example: Role-Based Authorization

```
public class ProductsController : ApiController
{
    [Authorize(Roles = "Admin")]
    [HttpPost]
    public IHttpActionResult AddProduct([FromBody] Product product)
    {
        // Only users with the "Admin" role can access this resource
        productService.AddProduct(product);
        return Created("api/products", product); // 201 Created
    }
}
```

In this example, the [Authorize(Roles = "Admin")] attribute ensures that only users with the "Admin" role can access the AddProduct action method.

These examples demonstrate the versatility of authentication and authorization mechanisms in .NET Framework 4.8 Web API. Depending on your project requirements and security policies, you can choose the most suitable approach or combine multiple methods for a comprehensive security strategy.

### 4.3 Exception Handling

Proper exception handling improves error reporting and enhances the security of your Web API. Implement global exception filters to handle unexpected errors gracefully.

Example: Global Exception Filter

```
public class GlobalExceptionFilter : ExceptionFilterAttribute
{
    public override void OnException(HttpActionExecutedContext context)
    {
        var exception = context.Exception;

        // Log the exception or take necessary actions

        context.Response = new
HttpResponseMessage(HttpStatusCode.InternalServerError)
        {
            Content = new StringContent("An unexpected error occurred."),
            ReasonPhrase = "Internal Server Error"
        };
    }
}
```

In this example, unexpected exceptions are caught globally, and an internal server error response is returned with a meaningful message.

These security measures, including CORS configuration, authentication with JWT, and global exception handling, contribute to building a robust and secure .NET Framework 4.8 Web API. Customization and additional security layers can be added based on specific project requirements.

## 4. HTTP Caching in .NET Framework 4.8 Web API

HTTP caching plays a crucial role in optimizing the performance of a Web API by reducing the need to repeatedly fetch resources from the server. Properly configured caching mechanisms enhance response times and minimize server load. In this section, we'll explore HTTP caching strategies and how to implement them in a .NET Framework 4.8 Web API.

### 4.1 Overview of HTTP Caching

HTTP caching involves storing copies of resources, such as images or API responses, either on the client side or intermediary servers (e.g., proxy servers). When a client requests a resource, the server checks if the resource is already cached on the client or any intermediary server. If the resource is present and still valid (not expired), the server responds with a lightweight "Not

Modified" (304) status, and the client uses its cached copy, avoiding the need to transfer the entire resource again.

## 4.2 Cache-Control Header

The Cache-Control header is a powerful tool for controlling caching behavior. It provides directives that instruct how caches should store, request, and respond to resources.

Example: Cache-Control Header

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

- **max-age**: Specifies the maximum amount of time a resource is considered fresh in seconds.
- **public**: Indicates that the response may be cached by any cache, even if it's normally non-cacheable.

## 4.3 Implementing HTTP Caching in .NET Framework 4.8 Web API

### 4.3.1 Caching Responses

In your Web API controllers, you can use the CacheControl attribute to control caching behavior.

Example: Caching Responses in Web API

```
[HttpGet]
[CacheControl(MaxAge = 3600, SharedMaxAge = 3600)]
public IHttpActionResult Get()
{
    // Retrieve and return data
    var data = GetData();

    return Ok(data);
}
```

In this example, the CacheControl attribute is applied to the Get action method, specifying a max-age of 3600 seconds (1 hour) and a shared-max-age of 3600 seconds.

### 4.3.2 Cache Validation

To implement cache validation, use the CacheCow.Server library. It allows you to leverage conditional requests using ETag and Last-Modified headers.

Example: Cache Validation in Web API

```
[HttpGet]
[HttpCacheValidation(GenerateEtag = true, LastModified = true)]
public IHttpActionResult Get(int id)
{
```

```
    var data = GetDataById(id);

    // Check if data is modified
    if (Request.Headers.IfNoneMatch.Any() &&
Request.Headers.IfNoneMatch.First().Tag == data.Etag)
    {
        return StatusCode(HttpStatusCode.NotModified);
    }

    return Ok(data);
}
```

In this example, the HttpCacheValidation attribute is applied, enabling ETag and Last-Modified header support. The action method checks these headers to determine if the resource has been modified.

### 4.4 Additional Considerations

**Validation Mechanisms**: Leverage ETag and Last-Modified headers for efficient cache validation.
Cache Invalidation: Implement cache invalidation strategies when data changes to ensure clients receive up-to-date information.

## 5. API Versioning in .NET Framework 4.8 Web API

API versioning is crucial for managing changes to your Web API over time while ensuring backward compatibility. This section covers different strategies for versioning your API in a .NET Framework 4.8 Web API project.

### 5.1 Why API Versioning?

As your API evolves, you may introduce changes that could potentially break existing clients. API versioning allows you to introduce new features, modify existing ones, or deprecate certain functionalities in a controlled manner. This helps to avoid disruptions for existing users while enabling them to migrate to newer versions when ready.

### 5.2 API Versioning Strategies

#### 5.2.1 URI Versioning

In URI versioning, the version information is included directly in the URI.
Example: URI Versioning

```
https://api.example.com/v1/products
https://api.example.com/v2/products
```

#### 5.2.2 Query Parameter Versioning

In query parameter versioning, the version information is included as a parameter in the query string.

Example: Query Parameter Versioning

```
https://api.example.com/products?version=1
https://api.example.com/products?version=2
```

#### 5.2.3 Header Versioning

In header versioning, the version information is included in the request headers.

Example: Header Versioning

```
GET /products HTTP/1.1
Host: api.example.com
Api-Version: 1
```

#### 5.2.4 Media Type Versioning

In media type versioning, the version information is included in the Accept header.

Example: Media Type Versioning

```
GET /products HTTP/1.1
Host: api.example.com
Accept: application/json;v=1
```

### 5.3 Implementing API Versioning in .NET Framework 4.8 Web API

#### 5.3.1 Versioning with Route Constraints

Example: Route Constraints Versioning

```
config.Routes.MapHttpRoute(
    name: "DefaultApi",
    routeTemplate: "api/v{version}/{controller}/{id}",
    defaults: new { id = RouteParameter.Optional }
);
```

In this example, the API version is specified in the route template, allowing for version-specific controllers.

#### 5.3.2 Versioning with Query Parameters

Example: Query Parameter Versioning

```
[HttpGet]
[Route("products")]
```

```
public IHttpActionResult GetProducts([FromUri] int version)
{
    // Implement version-specific logic
}
```
In this example, the API version is extracted from the query parameter in the GetProducts action method.

### 5.3.3 Versioning with Headers

Example: Header Versioning

```
[HttpGet]
[Route("products")]
public IHttpActionResult GetProducts()
{
    var version =
Request.Headers.GetValues("Api-Version").FirstOrDefault();
    // Implement version-specific logic
}
```
In this example, the API version is extracted from the Api-Version header in the GetProducts action method.

### 5.3.4 Versioning with Media Type

Example: Media Type Versioning

```
[HttpGet]
[Route("products")]
public IHttpActionResult GetProducts()
{
    var acceptHeader = Request.Headers.Accept.FirstOrDefault();
    var version = acceptHeader?.Parameters.FirstOrDefault(p => p.Name ==
"v")?.Value;
    // Implement version-specific logic
}
```
In this example, the API version is extracted from the Accept header in the GetProducts action method.

## 6. Use of Swagger in .NET Framework 4.8 Web API

Swagger is a powerful tool that simplifies the documentation and testing of APIs. It allows developers to visualize and interact with the endpoints of a Web API, making it easier to understand, test, and consume. In this section, we'll explore how to integrate Swagger into a .NET Framework 4.8 Web API project.

## 6.1 What is Swagger?

Swagger is an open-source framework for designing, building, documenting, and consuming RESTful APIs. It provides a standard way to describe RESTful APIs using a common language and offers tools for generating interactive documentation, client SDK generation, and API testing.

## 6.2 Integrating Swagger into .NET Framework 4.8 Web API

### 6.2.1 Installing Swashbuckle NuGet Package

To integrate Swagger into your .NET Framework 4.8 Web API project, you can use the Swashbuckle NuGet package. Swashbuckle is a popular library that seamlessly integrates Swagger into ASP.NET Web API projects.

Example: Installing Swashbuckle

```
Install-Package Swashbuckle -Version 5.6.3
```

### 6.2.2 Configuring Swagger in Global.asax

In your Global.asax.cs file, configure Swagger in the Application_Start method.

Example: Configuring Swagger

```
protected void Application_Start(object sender, EventArgs e)
{
    GlobalConfiguration.Configure(WebApiConfig.Register);

    // Enable Swagger
    GlobalConfiguration.Configuration
        .EnableSwagger(c =>
        {
            c.SingleApiVersion("v1", "Your API Name");
            c.IncludeXmlComments(GetXmlCommentsPath());
        })
        .EnableSwaggerUi(c => { });
}

private static string GetXmlCommentsPath()
{
    return System.AppDomain.CurrentDomain.BaseDirectory +
@"\YourApiName.xml";
}
```

### 6.2.3 Adding XML Documentation Comments

Swagger can use XML documentation comments to enhance the generated documentation. Ensure that your project is set to generate XML documentation.

- Right-click on your project in Solution Explorer.
- Select "Properties."
- Go to the "Build" tab.
- Check the "XML documentation file" option and provide a path (e.g., bin\YourApiName.xml).

### 6.2.4 Running and Accessing Swagger

After configuring Swagger, run your Web API project, and navigate to the Swagger UI by appending /swagger to your API's base URL.

Example: Swagger UI URL

```
http://localhost:YourPort/swagger
```

## 6.3 Swagger Annotations

Enhance your Web API documentation by using Swagger annotations to provide additional information about your API, such as response types, parameters, and authentication details.

Example: Swagger Annotations in Web API

```
[HttpGet]
[Route("products/{id}")]
[SwaggerResponse(HttpStatusCode.OK, Type = typeof(Product))]
[SwaggerResponse(HttpStatusCode.NotFound, Description = "Product not
found")]
public IHttpActionResult GetProductById(int id)
{
    var product = productService.GetProductById(id);

    if (product == null)
    {
        return NotFound(); // 404 Not Found
    }

    return Ok(product); // 200 OK
}
```

In this example, the SwaggerResponse attribute is used to specify the expected HTTP responses for the GetProductById action method.

## 6.4 Additional Swagger Features

Swagger provides additional features such as API versioning support, customizing the Swagger UI, and securing the Swagger documentation endpoint. Explore these features to tailor Swagger to your project's needs.

By integrating Swagger into your .NET Framework 4.8 Web API project, you enhance the development experience by providing a self-documenting and interactive interface for exploring and testing your API endpoints.

# 7. Use of POSTMAN in .NET Framework 4.8 Web API

POSTMAN is a popular API development and testing tool that simplifies the process of sending HTTP requests and inspecting responses. It allows developers to create, share, test, and document APIs efficiently. In this section, we'll explore how to use POSTMAN to interact with a .NET Framework 4.8 Web API.

## 7.1 What is POSTMAN?

POSTMAN is a collaboration platform for API development. It provides a user-friendly interface for building, testing, and documenting APIs. POSTMAN supports various HTTP methods, authentication types, and request/response formats, making it a versatile tool for API development and testing.

## 7.2 Using POSTMAN with .NET Framework 4.8 Web API

### 7.2.1 Installation

Download and install POSTMAN from the official website.
Launch POSTMAN after installation.

### 7.2.2 Making Requests

#### 7.2.2.1 Sending GET Requests

- Open POSTMAN.
- In the URL field, enter the endpoint URL for a GET request (e.g., http://localhost:YourPort/api/products).
- Choose the HTTP method as GET.
- Click the "Send" button to execute the request.
- View the response in the lower part of the POSTMAN window.

#### 7.2.2.2 Sending POST Requests

- Open POSTMAN.
- In the URL field, enter the endpoint URL for a POST request (e.g., http://localhost:YourPort/api/products).
- Choose the HTTP method as POST.
- Go to the "Body" tab, select "raw," and choose the request format (e.g., JSON).

- Enter the request body.
- Click the "Send" button to execute the request.
- View the response in the lower part of the POSTMAN window.

#### 7.2.2.3 Sending Other Requests

- POSTMAN supports various HTTP methods (GET, POST, PUT, DELETE, etc.), and you can set headers, authentication, and parameters easily.

### 7.2.3 Saving and Organizing Requests

- After creating a request, you can save it for future use.
- Click the "Save" button, enter a name, and choose a collection to save the request.
- Organize your requests into collections for better management.

### 7.2.4 Environment Variables

POSTMAN allows you to define environment variables to streamline testing across different environments.

- Click on the gear icon in the top-right corner.
- Choose "Manage Environments."
- Add a new environment and define variables (e.g., base URL).
- 7.2.5 Testing and Automation
- POSTMAN supports writing tests using JavaScript. You can write scripts to automate testing and validation.
-
- In a request, go to the "Tests" tab.
- Write JavaScript code to perform tests on the response.

```
pm.test("Status code is 200", function () {
    pm.response.to.have.status(200);
});

pm.test("Response should be valid JSON", function () {
    pm.response.to.be.json;
});
```

# 8. Deployment of .NET Framework 4.8 Web API

Deploying a .NET Framework 4.8 Web API involves making the application accessible for users or other systems. This section covers the basic steps for deploying a .NET Framework 4.8 Web API to a hosting environment.

## 8.1 Compilation and Publishing

Before deploying, you need to compile and publish your .NET Framework 4.8 Web API project. The compilation process generates the necessary binaries, and publishing prepares the

application for deployment.

### 8.1.1 Compilation

In Visual Studio:

- 
- Open your .NET Framework 4.8 Web API project.
- Build the project to ensure there are no compilation errors.
- Right-click on the project in Solution Explorer.
- Select "Build."

### 8.1.2 Publishing

- Right-click on the project in Solution Explorer.
- Select "Publish."
- Choose a publishing target (e.g., File System, FTP, Azure, etc.).
- Configure the publishing settings, including the target directory or service credentials.
- Click "Publish" to generate the deployment artifacts.

## 8.2 Hosting Environment

Choose a hosting environment that supports .NET Framework 4.8. Common hosting options include:

IIS (Internet Information Services): Deploying to IIS is a common choice for hosting .NET Framework applications. You can create an IIS site and deploy your application to it.

Azure App Service: If you prefer a Platform-as-a-Service (PaaS) solution, Azure App Service supports hosting .NET Framework applications. You can deploy directly from Visual Studio or use continuous integration.

## 8.3 Database and Configuration

Ensure that the database and configuration settings are correctly configured for the production environment. Update connection strings, app settings, and other configuration parameters to match the production environment.

## 8.4 Securing the Deployment

Consider securing your deployed Web API by:

Using HTTPS: Configure your hosting environment to use HTTPS to encrypt data in transit.

Authentication and Authorization: Implement proper authentication and authorization mechanisms to control access to your API endpoints.

## 8.5 Monitoring and Logging

Implement monitoring and logging to track the performance and health of your deployed Web API. Use tools like Application Insights, ELK Stack, or custom logging solutions to capture relevant logs and metrics.

## 8.6 Continuous Integration and Deployment (CI/CD)

Consider setting up a CI/CD pipeline to automate the deployment process. This ensures a smooth and consistent deployment experience for every code change.

## 8.7 Post-Deployment Testing

After deployment, perform thorough testing to ensure that the deployed Web API functions as expected in the production environment. Test various scenarios, including different API endpoints, authentication, and error handling.

## 8.8 Rollback Plan

Always have a rollback plan in place in case issues arise during or after deployment. This may include having a backup of the previous version, using feature toggles, or having a process to quickly revert to a stable state.

## 8.9 Documentation

Update or create documentation for the deployed Web API, including API documentation for users and operational documentation for administrators.