## ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION)

# DEPLOYMENT TECHNIQUES

## DEPLOYING ASP.NET CORE APPLICATIONS TO IIS, AZURE, AND AWS

Deployment is an essential step in the software development lifecycle that allows developers to move applications from development to production. ASP.Net Core applications can be deployed to different hosting environments such as IIS (Internet Information Services), Azure, and AWS (Amazon Web Services). Each environment has its unique advantages and deployment strategies, but understanding how to deploy to each one is crucial for building scalable, secure, and performant applications.

## DEPLOYING TO IIS (INTERNET INFORMATION SERVICES)

IIS is a popular web server for hosting .NET applications on Windows servers. Deploying an ASP.Net Core application to IIS involves several steps, including configuring IIS, setting up the application pool, and configuring the server for ASP.Net Core. One of the benefits of IIS is its powerful management tools, which make it easier to manage and monitor applications.

1. **Install the .NET Core Hosting Bundle**: First, you need to install the .NET Core Hosting Bundle on the server where IIS is running. The Hosting Bundle includes the .NET Core runtime and the ASP.Net Core module, which allows IIS to forward requests to the Kestrel server (the internal web server used by ASP.Net Core).

2. **Publish the Application**: In Visual Studio, right-click the project, select **Publish**, and choose the **Folder** publish target. This will create the necessary files for deployment. You can choose a local folder on your machine to publish the application.

3. **Set Up IIS**:
   - Open **IIS Manager**.
   - Add a new **Website** or **Application**.
   - Point the **Physical Path** to the folder where you published the application.
   - Create a new **Application Pool** and configure it to use **No Managed Code** (since ASP.Net Core runs on Kestrel and does not rely on the IIS managed runtime).
4. **Configure the Web.config**: ASP.Net Core requires a web.config file to properly route requests between IIS and the Kestrel server. This file is generated automatically when you publish the application, but you can manually edit it if necessary.

Example of web.config:

```
<configuration>
  <system.webServer>
    <handlers>
      <add name="aspNetCore" path="*" verb="*" modules="AspNetCoreModuleV2" resourceType="Unspecified" requireAccess="None" />
    </handlers>
    <aspNetCore processPath="dotnet" arguments=".\MyApp.dll" stdoutLogEnabled="false" stdoutLogFile="\\?\%temp%\stdout" />
  </system.webServer>
</configuration>
```

After configuring the application, IIS will handle incoming HTTP requests, passing them to Kestrel for processing.

**Deploying to Azure**

Azure is a popular cloud computing platform, and deploying ASP.Net Core applications to Azure is a straightforward process that provides scalability, reliability, and various integration services. Microsoft Azure provides several deployment options, such as **Azure App Service**, **Azure Virtual Machines**, and **Azure Kubernetes Service** (AKS), with App Service being the most common choice for ASP.Net Core applications.

1. **Create an Azure App Service**: In the Azure portal, create a new **App Service**. Select your subscription, resource group, and app service plan (pricing tier). Azure will handle scaling and maintenance of the environment.

2. **Publish the Application**: From Visual Studio, you can publish the ASP.Net Core application directly to Azure App Service by selecting **Publish** and choosing **Azure** as the target. You'll need to sign in with your Azure account and select the App Service you created.

3. **Set Up Deployment Slot**: For better control over deployments, you can create **deployment slots**. Slots allow you to deploy your application to a staging environment first, and then swap it with the production environment once you are ready.

4. **Configure Continuous Deployment**: Azure App Service can be integrated with **GitHub** or **Azure DevOps** for continuous deployment. This allows automatic deployment every time you push changes to a specific branch.

## DEPLOYING TO AWS

Amazon Web Services (AWS) offers a variety of services for deploying ASP.Net Core applications, including EC2 (Elastic Compute Cloud), Elastic Beanstalk, and Amazon ECS (Elastic Container Service). For most simple applications, **AWS Elastic Beanstalk** provides a PaaS solution for ASP.Net Core applications, managing infrastructure and scaling automatically.

1. **Set Up Elastic Beanstalk**:
   - In the AWS Management Console, create an **Elastic Beanstalk** environment.
   - Choose **.NET Core on Windows Server** as the platform.
   - AWS will manage the environment, including load balancing, auto-scaling, and application health monitoring.

2. **Publish the Application**: From Visual Studio, use the **AWS Toolkit for Visual Studio** to deploy your ASP.Net Core application to Elastic Beanstalk. You can also use the **AWS CLI** or **Elastic Beanstalk CLI** to deploy manually.

3. **Configuration and Monitoring**: AWS provides tools such as **CloudWatch** for monitoring application performance and **CloudFormation** for automating infrastructure setup.

---

## CI/CD Pipeline using GitHub and Azure DevOps

A CI/CD (Continuous Integration and Continuous Deployment) pipeline automates the process of building, testing, and deploying applications, significantly improving development efficiency and ensuring reliable, consistent deployments. Both **GitHub Actions** and **Azure DevOps** offer powerful tools for setting up CI/CD pipelines.

### CI/CD with GitHub Actions

GitHub Actions is a feature that allows you to automate workflows directly in your GitHub repository. These workflows can include tasks such as building the application, running tests, and deploying it to your target environment.

1. **Set Up GitHub Actions**:
   - In your GitHub repository, go to the **Actions** tab and select **New Workflow**.
   - GitHub provides templates for various workflows, including **.NET Core** applications.
2. **Create a Workflow File**:
   - In the .github/workflows directory of your repository, create a YAML file (e.g., ci-cd-pipeline.yml).
   - Define steps such as installing dependencies, building the project, running tests, and deploying to Azure or AWS.

Example of a GitHub Actions pipeline for an ASP.Net Core app:

```
name: CI/CD Pipeline for ASP.Net Core App


on:

  push:
    branches:
      - main

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v2
      - name: Set up .NET
        uses: actions/setup-dotnet@v1
        with:
          dotnet-version: '5.0'
      - name: Restore dependencies
        run: dotnet restore
      - name: Build
        run: dotnet build --configuration Release
      - name: Test
        run: dotnet test
      - name: Publish
        run: dotnet publish --configuration Release --output ./publish
      - name: Deploy to Azure
        uses: azure/webapps-deploy@v2
        with:
```

```
        app-name: your-azure-app-service-name

        publish-profile: ${{ secrets.AZURE_PUBLISH_PROFILE }}

        package: ./publish
```

**CI/CD with Azure DevOps**

Azure DevOps provides a set of tools for automating the CI/CD pipeline, including **Azure Pipelines**, which supports a variety of languages and platforms, including .NET Core.

1. **Set Up a Pipeline**:
   - Create a new **Pipeline** in Azure DevOps.
   - Choose **Azure Repos Git** (or GitHub) as the source, and select the repository.
   - Define pipeline stages for building, testing, and deploying the application.
2. **YAML Pipeline Configuration**: Azure DevOps allows you to define your CI/CD pipeline using YAML files, similar to GitHub Actions.

Example of an Azure DevOps pipeline for an ASP.Net Core app:

```
trigger:

- main



pool:

  vmImage: 'windows-latest'



steps:

- task: UseDotNet@2

  inputs:

    packageType: 'sdk'
    version: '5.x'
    installationPath: $(Agent.ToolsDirectory)/dotnet
- task: Restore@2
  inputs:
    command: restore
    projects: '**/*.csproj'
- task: Build@1
  inputs:
    command: build
    projects: '**/*.csproj'
- task: Test@2
  inputs:
    command: test
    projects: '**/*.csproj'
- task: Publish@1
  inputs:
    command: publish
    publishWebProjects: true
    projects: '**/*.csproj'
    arguments: '-configuration Release -output $(Build.ArtifactStagingDirectory)'
    zipAfterPublish: true
- task: AzureWebApp@1
  inputs:
    azureSubscription: 'your-azure-subscription'
    appName: 'your-app-service-name'

    package: '$(Build.ArtifactStagingDirectory)/**/*.zip'
```

---

## EXERCISES AND CASE STUDY

**Exercise:**

1. **Deploy an ASP.Net Core App to IIS**:
   - Deploy your ASP.Net Core application to a local or remote IIS server and verify that it is running successfully.
2. **Create a CI/CD Pipeline using GitHub Actions**:
   - Set up a GitHub repository and create a CI/CD pipeline for your ASP.Net Core application using GitHub Actions to build and deploy the app to Azure.

**Case Study:**

Company XYZ is building a cloud-native application using ASP.Net Core. The company needs to deploy the application to multiple environments, including IIS for internal use, Azure for cloud deployment, and AWS for scalability. Additionally, the company wants to automate the build, test, and deployment process using GitHub Actions and Azure DevOps.

Your task is to:

1. Set up deployment pipelines for IIS, Azure, and AWS.
2. Automate the CI/CD process using GitHub Actions and Azure DevOps.
3. Ensure the application is properly secured, scalable, and maintainable across all deployment environments.

# Application Monitoring and Maintenance

## Logging, Error Tracking, and Performance Monitoring

Effective application monitoring and maintenance are critical to ensuring the reliability, performance, and security of web applications in production. Monitoring allows developers to detect issues, optimize performance, and maintain the overall health of the system. In this chapter, we will discuss **logging**, **error tracking**, and **performance monitoring**—all of which are vital for keeping an application running smoothly.

**Logging**

**Logging** is a mechanism for recording information about an application's operation. This information helps developers and operations teams understand what happens in the application during runtime, allowing them to troubleshoot issues, monitor application behavior, and improve performance. Logs can capture a wide range of information, such as system events, user interactions, exceptions, and performance metrics.

In ASP.Net Core, the **ILogger** interface is used for logging. By default, ASP.Net Core provides logging providers for output to the console, debug window, and files. However, you can configure logging to integrate with third-party logging solutions such as **Serilog**, **NLog**, or **Log4Net**.

Here's an example of logging in an ASP.Net Core application:

```
public class ProductService {

  private readonly ILogger<ProductService> _logger;



  public ProductService(ILogger<ProductService> logger) {

    _logger = logger;

  }



  public void AddProduct(string productName) {
    _logger.LogInformation($"Adding product: {productName}");

    try {

      // Code to add product

    } catch (Exception ex) {

      _logger.LogError(ex, "Error occurred while adding product");

    }

  }

}
```

In this example, we use LogInformation to log the action of adding a product and LogError to capture any errors that may occur during the process. This allows for easy tracking of what is happening inside the application.

**Error Tracking**

Error tracking is the process of logging and monitoring errors that occur in production. This allows developers to identify bugs and other issues in real time, so they can be fixed quickly. While logging captures detailed information about exceptions, **error tracking** tools like **Sentry**, **Raygun**, and **Rollbar** provide enhanced capabilities for detecting, diagnosing, and resolving errors in production environments.

ASP.Net Core has built-in support for **exception handling** and logging, but integrating error tracking tools offers more advanced features such as:

- **Real-time error reporting**: Instant notifications when an error occurs.
- **Error aggregation**: Grouping similar errors to avoid cluttering the logs.
- **Stack traces**: Detailed information about the error and the sequence of function calls leading to it.

Here's an example of integrating **Sentry** for error tracking:

```
public void ConfigureServices(IServiceCollection services) {

  services.AddSentry(options => {

    options.Dsn = "your_sentry_dsn";
    options.TracesSampleRate = 1.0;  // 100% of transactions are sent to Sentry
  });
}
```

By using tools like Sentry, you can quickly identify issues in your application, track them over time, and ensure that the development team is alerted to high-priority errors.

**Performance Monitoring**
Performance monitoring is crucial for ensuring that an application is responsive and performs optimally under load. Without proper monitoring, users may experience slow load times, timeouts, or degraded service, which can result in poor user experience and loss of business.

**Application performance monitoring (APM)** tools, such as **New Relic**, **AppDynamics**, and **Azure Application Insights**, are widely used to track performance metrics like:

- **Response times**: The time it takes for the server to respond to requests.
- **Throughput**: The number of requests handled by the application within a given time period.
- **Error rates**: The frequency of errors occurring in the system.
- **Resource usage**: CPU, memory, and network usage.

In ASP.Net Core, you can integrate performance monitoring using built-in tools like **Application Insights**:

```
public void ConfigureServices(IServiceCollection services) {

services.AddApplicationInsightsTelemetry(Configuration["ApplicationInsights:InstrumentationKey"]);

}
```

This setup enables real-time performance metrics, and you can view the performance data in the Azure portal. Performance monitoring tools also offer deeper insights such as transaction tracing and slow request analysis.

## Key Benefits of Application Monitoring:

- **Quick Issue Detection**: By tracking errors and performance metrics, you can identify issues in real-time and address them before they affect a large number of users.
- **Proactive Maintenance**: Regular monitoring can help identify potential bottlenecks or areas for optimization, allowing you to improve the system before performance degrades.
- **Optimized User Experience**: By ensuring the application is running efficiently and without errors, you improve user satisfaction and retention.

## UPDATING AND MAINTAINING APPLICATIONS IN PRODUCTION

Updating and maintaining applications in production is a delicate process that requires careful planning and execution to minimize downtime, avoid data loss, and ensure seamless user experience. This involves handling updates, patches, and bug fixes while keeping the system stable and secure.

**Continuous Integration and Continuous Deployment (CI/CD)**
To ensure that updates to an application are deployed efficiently and consistently, many development teams use **CI/CD** pipelines. CI/CD automates the process of testing, building, and deploying code changes, reducing the risk of human error and improving the speed of delivering new features or fixes.

In ASP.Net Core, CI/CD can be set up with tools like **Azure DevOps**, **GitHub Actions**, or **Jenkins**. Here's an example of a simple Azure DevOps pipeline for deploying an ASP.Net Core application:

1. **Build Pipeline**: This pipeline automatically builds the application whenever changes are pushed to the repository.
2. **Release Pipeline**: Once the build is complete, this pipeline deploys the application to different environments (e.g., staging, production).

A typical YAML file for a GitHub Actions pipeline might look like:

```
name: ASP.Net Core CI/CD Pipeline


on:

  push:

    branches:
```

```
    - main


jobs:

  build:

    runs-on: ubuntu-latest

    steps:

    - uses: actions/checkout@v2

    - name: Set up .NET Core

      uses: actions/setup-dotnet@v1

      with:

        dotnet-version: '3.1.x'

    - name: Build

      run: dotnet build --configuration Release

    - name: Publish

      run: dotnet publish --configuration Release --output ./out

    - name: Deploy to Azure

      run: az webapp deploy --name <your-app-name> --resource-group <your-
resource-group> --src-path ./out
```

This CI/CD pipeline helps automate the entire deployment process, ensuring that updates are tested, built, and deployed efficiently.

**Blue-Green Deployment and Canary Releases**

To ensure zero downtime during production updates, **blue-green deployment** and **canary releases** are commonly used strategies:

- **Blue-Green Deployment**: This strategy involves maintaining two identical environments: one (the "blue" environment) is the live production environment, while the other (the "green" environment) contains the updated version. Traffic is routed to the blue environment until the green environment is fully tested and stable, at which point traffic is switched to the green environment.

- **Canary Releases**: In this approach, new updates are rolled out to a small subset of users first (the "canaries"). If the release is successful, it is gradually rolled out to the entire user base. This allows for easy rollback if an issue arises.

**Database Maintenance**
Maintaining databases in production can involve tasks such as schema changes, data migrations, and performance tuning. **Entity Framework Core** migrations are often used to apply database changes in a controlled manner. For example:

```
dotnet ef migrations add AddNewFieldToProduct

dotnet ef database update
```

This ensures that database changes are applied incrementally, and the application remains consistent with the database schema.

## Key Considerations for Maintaining Applications in Production:

- **Downtime Management**: Plan updates to minimize downtime and ensure high availability. Consider rolling updates or blue-green deployments.
- **Backup and Recovery**: Regular backups are crucial to avoid data loss during updates or unexpected failures.
- **Security Patches**: Apply security patches promptly to mitigate vulnerabilities and ensure the safety of user data.
- **Monitoring and Alerts**: Continuously monitor the application for issues post-deployment. Set up alerts for abnormal performance metrics or error rates.

## Exercise

1. **Implement Application Monitoring**: Set up logging and error tracking for an ASP.Net Core application using **Serilog** and **Sentry**. Monitor errors and performance metrics in real time.
2. **Deploy with CI/CD**: Set up a CI/CD pipeline for an ASP.Net Core application using GitHub Actions or Azure DevOps. Implement a simple automated deployment to a staging environment.

3. **Blue-Green Deployment**: Implement a blue-green deployment strategy for updating a web application. Use two environments (blue and green) to deploy new versions with zero downtime.

## Case Study: E-Commerce Application Maintenance

In this case study, you will manage and maintain an **E-Commerce Application** in a production environment. The application supports high traffic and frequently requires updates, bug fixes, and performance optimizations.

1. **Implement CI/CD**: Automate the deployment process for rolling out updates and fixes to the production environment using CI/CD pipelines.
2. **Monitor Performance**: Use performance monitoring tools like **Application Insights** or **New Relic** to track the application's health and address issues such as slow response times or errors in real-time.
3. **Database Migration**: Use **Entity Framework Core** migrations to update the product catalog schema without affecting users, ensuring that the database remains consistent with application updates.

# FINAL PROJECT AND REVIEW

## WORK ON A REAL-WORLD PROJECT, INTEGRATING CONCEPTS LEARNED IN THE COURSE

A final project is an essential component of any course, providing an opportunity for students to apply the concepts and skills they have learned throughout the course in a real-world scenario. This project acts as a culmination of your learning experience, where you will work independently or in teams to integrate various aspects of the curriculum. It not only helps reinforce theoretical knowledge but also allows students to demonstrate their ability to solve practical problems.

## PROJECT OVERVIEW

The primary objective of a final project is to create a fully functional application or system that solves a specific problem or meets a particular need. In the context of a software development course, the project could be an **ASP.Net Core Web Application**, a **RESTful API**, or a **cloud-based solution** involving technologies like **Azure**, **AWS**, or **Docker**. The final project allows students to showcase their understanding of various key concepts, such as:

- **Database management** (SQL Server, MySQL, etc.)
- **API development** (with authentication, CRUD operations)
- **Deployment** (on IIS, Azure, AWS)
- **CI/CD integration** (using tools like GitHub Actions or Azure DevOps)
- **Security and performance optimization** (JWT, caching, etc.)

**Project Execution**

To get started on the project, the following steps should be followed:

1. **Define the Problem or Objective**:
    - Identify a real-world problem that can be solved using the knowledge and tools you have learned. For example, you could build an e-commerce website, an inventory management system, or a customer relationship management (CRM) tool.

2. **Design the Solution**:
    - Create a blueprint of the solution by outlining the system architecture, databases, user interfaces, and key features. Plan how different modules and components will interact with each other.
3. **Develop the Application**:
    - Write the code, starting with setting up the backend (e.g., API development, database configuration). Then, move on to the frontend (UI development) and ensure seamless communication between the client and the server.
4. **Test the Application**:
    - Perform thorough testing on the system, including unit tests, integration tests, and user acceptance tests. Ensure that all components function as expected and handle edge cases gracefully.
5. **Deploy the Solution**:
    - Deploy the application to your chosen hosting environment, whether it's IIS, Azure, or AWS. Ensure that all configurations are correct, and the application is accessible and running smoothly.
6. **Optimize the Application**:
    - Apply performance optimization techniques, such as caching, efficient database queries, and optimizing server configurations. Ensure that the application is secure and can handle a high number of users.

---

### Presenting the Project and Feedback

Once the project is developed, the next step is to present it to an audience, typically your instructors, peers, or stakeholders. This is a critical phase of the project as it allows you to communicate your ideas, showcase the technical implementation, and receive feedback that can help improve your skills.

#### Presentation Structure

A well-structured presentation helps convey the key aspects of your project and demonstrates your understanding. Here's a suggested structure for your final project presentation:

1. **Introduction**:
    - Briefly introduce the project, its goals, and the problem it addresses. For example, "I've developed a web application for managing customer orders in a small business, which simplifies the order process and provides real-time inventory tracking."
2. **Technology Stack**:
    - Describe the tools and technologies you used to build the project, such as **ASP.Net Core**, **SQL Server**, **Azure**, **JWT**, or **Docker**. Explain why you chose these technologies and how they helped solve the problem.
3. **System Design**:
    - Walk through the architecture of the system, explaining how different components work together. For instance, "The backend uses a RESTful API to handle CRUD operations for customers and orders, while the frontend communicates with the API using AJAX requests."
4. **Key Features**:
    - Highlight the most important features of your project, such as user authentication, data validation, security, or performance optimizations. Show real-time demos of the functionality.
5. **Challenges Faced**:
    - Discuss any obstacles you faced during the project, such as debugging issues, handling database migrations, or deploying to the cloud. Explain how you overcame them.
6. **Future Improvements**:
    - Mention any potential improvements or features you would add if you had more time. This could include things like adding automated tests, improving UI design, or implementing machine learning for data analysis.
7. **Q&A Session**:
    - Encourage your audience to ask questions and provide feedback. This is a valuable opportunity to reflect on your work and discuss how it can be improved.

**Feedback**

Feedback from peers and instructors is crucial for personal growth and improving your work. When presenting the project, be open to constructive criticism, as it can help identify areas that need improvement or where you could take the project to the next level.

For example, feedback could include:

- Suggestions for better database schema design or more efficient queries.
- Improving security practices, such as enhancing password hashing or implementing role-based access control.
- Recommendations for better UI/UX design, ensuring that the application is user-friendly and intuitive.

---

## Exercises and Case Study

**Exercise:**

1. **Integrate Third-Party API**:
    - Extend your project by integrating a third-party API, such as a payment gateway (Stripe, PayPal) or a weather service. Demonstrate how the integration works during the project presentation.
2. **Optimize for Performance**:
    - Identify areas in your application that can be optimized for better performance. Use techniques like caching, query optimization, and background processing.
3. **Create a Comprehensive Documentation**:
    - Write detailed documentation for your project, including setup instructions, architecture diagrams, and API usage guides. This will not only help others understand your project but also enhance your portfolio.

**Case Study:**

Company XYZ is developing a customer-facing web application that allows users to place and track orders online. The application requires authentication for users, real-time order updates, and secure payment integration. The company has tasked you with building the application and deploying it to a cloud service.

Your task is to:

1. Build the backend API using **ASP.Net Core** to handle customer orders and payments.
2. Implement **JWT authentication** to secure user data and interactions.
3. Deploy the application to **Azure** and ensure it scales according to user demand.
4. Present the project to the company, demonstrating the app's features, and gather feedback for future improvements.

# Assignment Solution: Deploy a Fully Functional Web Application to a Cloud Service (Azure or AWS) and Monitor Its Performance

In this assignment, we will deploy a fully functional ASP.Net Core web application to a cloud service (either **Azure** or **AWS**), configure monitoring, and ensure its performance is tracked. This solution will guide you through setting up the application, deploying it to the cloud, and using cloud monitoring tools to keep track of performance metrics.

**Prerequisites:**

- An active **Azure** or **AWS** account.
- **Visual Studio** (or any other development environment) with **ASP.Net Core** installed.
- Basic knowledge of cloud computing and application deployment.

---

**Step-by-Step Guide**

## Step 1: Create a Fully Functional ASP.Net Core Web Application

1. **Create a New ASP.Net Core Application**:
    - Open **Visual Studio** and select **Create a New Project**.
    - Choose **ASP.Net Core Web Application**, and click **Next**.
    - Select **Web Application (Model-View-Controller)** and ensure **.NET Core** is selected.
    - Click **Create** to generate a basic MVC application.
2. **Build the Application**:

- Implement a simple feature for the application (e.g., a product catalog, user login, or basic API).
- Ensure the application runs correctly locally by clicking **Ctrl + F5**.

## STEP 2: SET UP CLOUD SERVICE (AZURE OR AWS)

**Azure Deployment**:

1. **Sign In to Azure**:
    - Go to the Azure portal and sign in with your Azure account.
2. **Create a New Web App**:
    - In the Azure portal, click on **Create a Resource**.
    - Under **Compute**, select **App Services**.
    - Fill in the necessary details:
        - **Subscription**: Choose your Azure subscription.
        - **Resource Group**: Either create a new resource group or use an existing one.
        - **App Name**: Give your application a unique name.
        - **Publish**: Select **Code**.
        - **Runtime Stack**: Choose **.NET Core**.
        - **Region**: Select a region close to your target audience.
    - Click **Review + Create**, then **Create**.
3. **Deploy the Application**:
    - Once the Web App is created, go to the **Overview** tab of your newly created web app.
    - Click **Deployment Center** to set up deployment options.
    - Choose **GitHub** or **Azure Repos** as your source for deployment.
        - If using **GitHub**:

            - Authenticate with your GitHub account.
            - Select the repository and branch to deploy from.
        - If using **Azure Repos**:
            - Connect to your Azure DevOps repository and configure your project.
    - After configuring the deployment source, click **Finish**. Azure will automatically deploy your application to the Web App.

**AWS Deployment**:

1. **Sign In to AWS**:
    - Go to the AWS Management Console and sign in.
2. **Create an Elastic Beanstalk Application**:
    - In the AWS Console, search for **Elastic Beanstalk** and select it.
    - Click on **Create New Application**.
    - Fill in the application name and environment name.
    - Select **.NET Core** as the platform, and choose the appropriate version (e.g., **.NET Core 3.1** or **.NET 5**).
    - Click **Create** to create the application environment.
3. **Deploy the Application**:
    - After the environment is created, click on **Upload and Deploy**.
    - Select the .zip file of your ASP.Net Core application or choose to deploy directly from your **GitHub** or **AWS CodePipeline** repository.
    - Once the upload is complete, click **Deploy** to start the process.

---

## STEP 3: CONFIGURE MONITORING AND PERFORMANCE TRACKING

**Azure Monitoring**:

1. **Enable Application Insights**:

- In the Azure portal, navigate to your **App Service** and click on **Application Insights**.
- Click **Turn on Application Insights** and select the region for monitoring.
- Application Insights will automatically start collecting telemetry data such as request times, exceptions, and performance metrics.
- Click **Apply** to enable monitoring.

2. **View Performance Metrics**:

   - In the Azure portal, go to **Application Insights** under your App Service.
   - You can view various metrics, including:
     - **Server response time**: View the average time it takes to process requests.
     - **Failure rates**: Track any failed requests or errors.
     - **Application Map**: View the performance and dependencies between your web app and services.
   - You can also set up **alerts** to notify you of performance issues (e.g., high response time or increased failure rates).

**AWS Monitoring**:

1. **Enable CloudWatch Monitoring**:

   - In the AWS Management Console, navigate to **Elastic Beanstalk** and select your environment.
   - Click on the **Monitoring** tab to view AWS CloudWatch metrics for your application.
   - You will see performance metrics such as:
     - **CPU Utilization**: Monitor how much CPU your application is using.
     - **Latency**: Track response times.

- **Request Count**: View how many requests are processed by the application.
- **Error Rates**: View 4xx and 5xx HTTP status codes.

  - Set up **CloudWatch Alarms** to alert you when certain thresholds are exceeded (e.g., CPU utilization above 80%).

2. **Set Up X-Ray for Tracing**:
   - AWS **X-Ray** is a service that helps trace requests as they travel through your application. You can enable **X-Ray** for your Elastic Beanstalk application:
     - In the **Elastic Beanstalk console**, go to your environment and select **Configuration**.
     - Under **Software**, select **Modify** and enable **X-Ray** for request tracing.
     - X-Ray will then capture detailed information about the requests, helping you identify bottlenecks or slow operations in the application.

---

## STEP 4: TESTING AND VERIFYING THE DEPLOYMENT

1. **Test the Deployed Application**:
   - Once the application is deployed, open the URL provided by **Azure** or **AWS** and verify that the application is running correctly in the cloud environment.
   - Perform a few actions in the application to ensure it responds correctly (e.g., navigating through pages, submitting forms, etc.).
2. **Monitor Performance**:
   - Use the **Azure Application Insights** or **AWS CloudWatch** dashboards to monitor the performance of your application. Ensure that the response times are within acceptable limits, and there are no unexpected error rates.
   - Check for any alerts or notifications triggered by performance issues.

## Step 5: Final Checks and Optimization

1. **Scaling Your Application**:
    - Both **Azure** and **AWS** allow you to scale your web application.
    - In **Azure**, you can scale your app manually or set up auto-scaling under **Scale Up** or **Scale Out** options.
    - In **AWS**, you can configure **Auto Scaling** under the Elastic Beanstalk environment settings.
2. **Review Logs**:
    - Check the logs for any errors or unusual activities in the production environment.
    - Use **Azure Log Stream** or **AWS CloudWatch Logs** to review the application logs.
3. **Security Considerations**:
    - Make sure to enable **HTTPS** to ensure secure communication between the client and the server.
    - Regularly update dependencies to avoid security vulnerabilities.

## Conclusion

By following these steps, you have successfully deployed a fully functional ASP.Net Core web application to **Azure** or **AWS**, configured monitoring, and tracked its performance in real time. This deployment setup helps you ensure that your application is performing well in production, with the ability to quickly identify and resolve any issues that arise.

## Exercise

1. **Deploy to AWS**: Create an Elastic Beanstalk environment for an ASP.Net Core web application and enable CloudWatch for performance monitoring.
2. **Deploy to Azure**: Set up Application Insights for an Azure Web App and monitor response times and failure rates.
3. **Performance Scaling**: Test the scaling options on both Azure and AWS by simulating high traffic and observing how the application adjusts its resources.