

Azure Monitor

Lab 1

Introduction of Application Insights

DISCLSIMER

© 2025 Microsoft Corporation. All rights reserved. Microsoft, Windows and other product names are or may be registered trademarks and/or trademarks in the U.S. and/or other countries.

The information herein is for informational purposes only and represents the current view of Microsoft Corporation or any Microsoft Group affiliate as of the date of this presentation. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information provided after the date of this presentation.

MICROSOFT MAKES NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, AS TO THE INFORMATION IN THIS PRESENTATION.

CONFIDENTIALITY

© 2025 Microsoft Corporation. All rights reserved. Any use or distribution of these materials without express authorization of Microsoft Corp. is strictly prohibited.

Contents

Lab 1 – Introduction of Application Insights (120 min)	4
Exercise 1- Create Application Insight (20min)	5
Module 1 – Create Application Insight in Azure	5
Module 2 – Get Connection String for App Insight.....	5
Module 3 – Instrument Your Application Insight for your Application	5
Module 4 – Configure App Insights in Program.cs.....	6
Module 5 – Verify App Insights for Your App.....	6
Module 6 – Deploy Your App to Azure	8
Summary.....	8
Exercise 2- Simulate and Troubleshoot Issues (30min)	10
Module 1 – Simple 404 (Not Found) Error	10
Module 2 – Create SQL Errors	11
Module 3 – Test your App and Verify	11
Module 4 – Troubleshoot Application Error	12
Module 5 – Implement Exception Handling	13
Module 6 – Run and Test Exception Handling	14
Module 7 (Optional) – Write Unit Testing	16
Summary.....	18
Exercise 3- Explore Metric and Availability (20min)	19
Module 1 – Create Metrics Chart(s)	19
Module 2 – Add Availability Test	19
Module 3 – View the Availability Test Results	20
Summary.....	20
Exercise 4- Configure Diagnostic Settings and Alerts (20min)	22
Module 1 – Enable Diagnostic Settings for Telemetry Data	22
Module 2 – Create an Alert for Failed Requests	22
Module 3 – Simulate Crash and Validate Alerts	23
Summary.....	24
Exercise 5- View Logs and Run Kusto Queries 30(min).....	25
Module 1 – Open Logs.....	25
Module 2 – Explorer and Modify Queries	26
Summary.....	26

Lab 1 – Introduction of Application Insights (120 min)

Scenario	<p>This lab introduces developers to Azure Application Insights for <i>monitoring and diagnosing</i> issues.</p> <p>We use the previously developed .NET 8 MVC application that connects to Azure SQL Database (AdventureWorks database).</p> <p>Developers will simulate real-world issues, explore telemetry data, set up alerts, and write simple Kusto queries.</p> <p>By the end of this lab, developers will be able to:</p> <ul style="list-style-type: none"> • Monitor application health and performance • Understand how to monitor app health and uptime • Diagnose issues using logs and metrics • Set up alerts for proactive monitoring • Use Kusto Query Language (KQL) to analyse telemetry data
Objective(s)	<ul style="list-style-type: none"> • Create Application Insight
Duration of Lab	<ul style="list-style-type: none"> • 120 min
Prerequisites	<ul style="list-style-type: none"> • Azure Portal • Visual Studio 2022 or Visual Studio Code • .NET MVC application is created and deployed to Azure (AdventureWorks application) • Backend Azure SQL DB is created and deployed to Azure (AdventureWorks DB)
Exercises	<ol style="list-style-type: none"> 1. Create Application Insight (20min) 2. Simulate and Troubleshoot Issues (30min) 3. Explorer Metrics and Availability (20min) 4. Configure Diagnostic Settings and Alerts (20min) 5. View Logs and Run Kusto Queries (30min)
Subscription	[Selected Subscription]
Resource Group	[Selected RG]
Navigation	<p>Throughout this Lab, we will open and use several Browser tabs for easy access.</p> <p>Until the end of the Lab, keep your Browser tabs open.</p>
References	<ul style="list-style-type: none"> • Introduction to Azure for developers Microsoft Learn

Naming Convention for Labs

For completing various labs during the workshops, we will use this naming convention. It is slightly different from Microsoft online guidance ([Define your naming convention - Cloud Adoption Framework | Microsoft Learn](#)).

The naming convention below is designed to group your Azure resources together for easy access.

[you name/initials]-[short name for Azure service]-[service description]

Exercise 1- Create Application Insight (20min)

Topics	<p>In this exercise, we will cover the following topics.</p> <ul style="list-style-type: none"> ➤ Create Application Insight ➤ Get Connection String for your App Insight ➤ Instrument Your App Insight for Your Application ➤ Configure App Insight ➤ Verify App Insights for Your App
Duration	<ul style="list-style-type: none"> • 20 min
Tool(s)	<ul style="list-style-type: none"> • Azure portal
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Create Application Insight in Azure

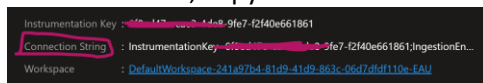
1. In the previous lab, you have created the Adventure Works web application with Azure SQL DB
2. If you haven't created Application Insight before, you need to create one for this lab
3. Go to **Azure portal** in a browser, create Application Insight
4. Enter the following details

Item	Description
Subscription	[your Subscription]
Resource Group	[your RG]
Name	[your short name]-appinsight-adventure-works-web
Region	Australiaeast
Workspace Details	[your Subscription]
Log Analytics Workspace	(leave as default)

5. Click **Review + create** and **Create**

Module 2 – Get Connection String for App Insight

1. Find your Application Insight for the Adventure Works application
2. If you created App Insight during App Service provisioning process before, you can find your associated App Insight from *App Service -> Application Insights in Monitoring*
3. Or you can search and find Application Insights as well
4. In the overview, copy the **Connection String** and paste it in Notepad



Module 3 – Instrument Your Application Insight for your Application

1. Open Visual Studio and open your previously created **AdventureWorks** application project
2. Open **Developer Command Prompt** in your project directory and run

```
dotnet add package Microsoft.ApplicationInsights.AspNetCore
```

3. Add the Connection string to *appsettings.json*
4. Open **appsettings.json**, and add the following *section* at the end

```
"ApplicationInsights": {
  "ConnectionString": "InstrumentationKey=xxxxxxx-xxxx-xxxx-xxxx-
xxxxxxxxxxx;IngestionEndpoint=https://<region>.applicationinsights.azure.com/"
}
```

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:mcaps-
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "AllowedHosts": "*",
  "ApplicationInsights": {
    "ConnectionString": "InstrumentationKey
  }
}
```

5. Replace the value with the one you've copied in Notepad

Module 4 – Configure App Insights in Program.cs

1. Open **Program.cs** and add the following code just before *"var app = builder.Build();"* statement:

```
// Configure Application Insights using options pattern
builder.Services.AddApplicationInsightsTelemetry(options =>
{
  options.ConnectionString = builder.Configuration["ApplicationInsights:ConnectionString"];
});
```

```
var builder = WebApplication.CreateBuilder(args);

// Add services to the container.
builder.Services.AddControllersWithViews();

builder.Services.AddDbContext<AdventureWorksContext>(options =>
options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultCon

// Configure Application Insights using options pattern
builder.Services.AddApplicationInsightsTelemetry(options =>
{
  options.ConnectionString = builder.Configuration["ApplicationInsights:Conn
});

var app = builder.Build();
```

2. Run your application locally (i.e. F5 or use the command 'dot run')

Module 5 – Verify App Insights for Your App

1. Navigate through your web application to generate telemetry
 - a. E.g. Select Privacy

- b. Navigate to /Products
 - c. Edit one of the products or add a new one
2. Go to Azure portal and select your App Insight
3. Go to **Application map** under the *Investigate* section
4. What do you see?



Application Insights – Application Map

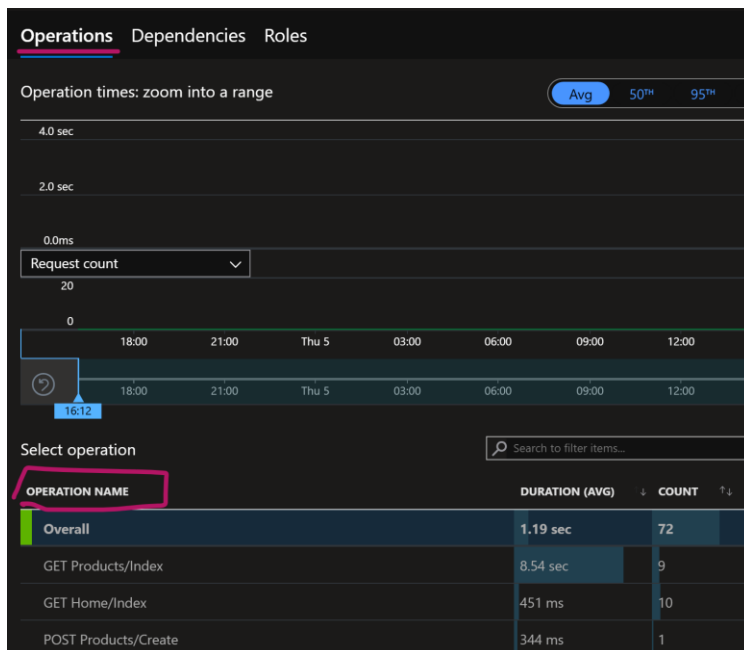
[Application map in Azure Application Insights - Azure Monitor | Microsoft Learn](#)

Developers use application maps to represent the logical structure of their distributed applications.

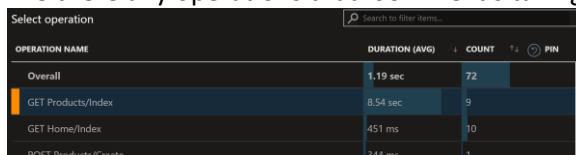
Circles (or nodes) on the map represent the components and *directional lines (connectors or edges)* show the HTTP calls from source nodes to target nodes.

Azure Monitor provides the Application map feature to help you quickly implement a map and spot performance bottlenecks or failure hotspots across all components.

5. Go to **Performance** under the Investigate section
6. In the *Operations* tab, you can see the list of executed operations for your application

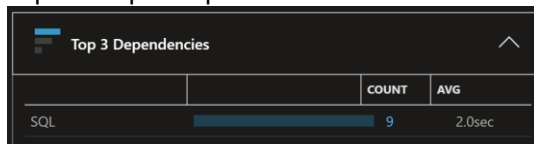


7. Are there any operations that look like it's taking a long time to complete?



OPERATION NAME	DURATION (AVG)	COUNT
Overall	1.19 sec	72
GET Products/Index	8.54 sec	9
GET Home/Index	451 ms	10
POST Products/Create	344 ms	1

8. Select the operation
9. What is the dependency(s) for this operation?
10. Expand Top 3 Dependencies section on the right pane



	COUNT	AVG
SQL	9	2.0sec

Module 6 – Deploy Your App to Azure

1. Deploy your app to Azure by using **Publish** method in VS
2. Once your application is deployed successfully, a new browser will be opened
3. Navigate around the web (e.g. navigate to /products, search, edit product etc...)
4. Go back to your App Insight in Azure portal
5. Go to **Live metrics** under the *Investigate* section



6. You will see real-time metrics for your web applications



Application Insights – Application Map

[Diagnose with live metrics - Application Insights - Azure Monitor | Microsoft Learn](#)

You can monitor web applications in real-time using **Live metrics** feature.

Live metrics view is helpful for validating a fix while it's released by watching performance and failure counts, and watch the effect of test/production loads and diagnose issues live.

- The data is getting displayed within one second, whilst Metrics in Log Analytics takes some time as the data is getting aggregated first (over minutes).
- The data is only available on the chart and then discarded. Not persisted at all
- Data is only streamed while the live metrics pane is open

7. Observe – *Request rate, Failures, Server response time and Dependency calls*

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Create Application Insight
- ✓ Instrument Application Insight for your Application
- ✓ Start observing your application activities in App Insight in Azure

- ✓ View the logical structure of the distributed components for your application (e.g. dependent services, endpoints etc...)

Exercise 2- Simulate and Troubleshoot Issues (30min)

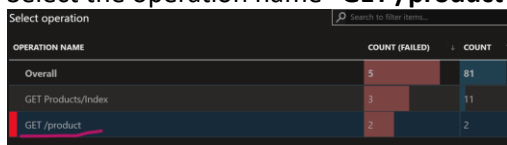
Topics	<p>In this exercise, we will cover the following topics.</p> <ul style="list-style-type: none"> ➤ Simple 404 (Not Found) Error ➤ Create SQL Errors ➤ Test your App and Verify ➤ Troubleshoot Application Error ➤ Implement Exception Handling ➤ Run and Test Exception Handling ➤ (Optional) – Write Unit Testing
Duration	<ul style="list-style-type: none"> • 30 Min
Tool(s)	<ul style="list-style-type: none"> • Azure portal • Visual Studio 2022
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Simple 404 (Not Found) Error

1. Run your app deployed in Azure (from App Service, click the URL in the Overview)
2. Navigate to **/product** in the address bar
3. What happens?
4. You will see the error like “...page can’t be found”



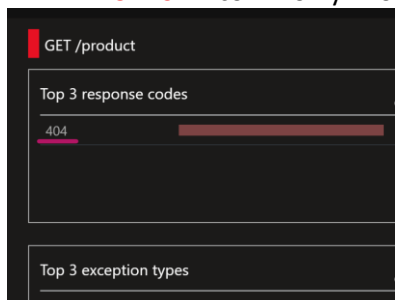
5. Go to your App Insight
6. Check if you have any failure(s) by selecting **Failures** in the *Investigate* section
7. You can see *Failed request* count and relevant *Operations*
8. Select the operation name “GET /product”



OPERATION NAME	COUNT (FAILED)	COUNT
Overall	5	81
GET Products/Index	3	11
GET /product	2	2

Note: The figure above is an example. You might not see all of failures (e.g. GET Product/Index)

9. What is the error response for this issue?
10. **HTTP Error 404** – commonly known as “*not found*” error



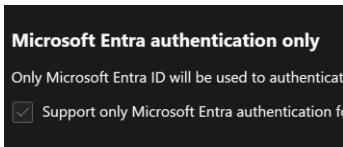
11. Your URL path is incorrect. We only have “/products” (plural) page available to list all the products available. Viewing a single product is to navigate to “/products/details/[id]”

Module 2 – Create SQL Errors

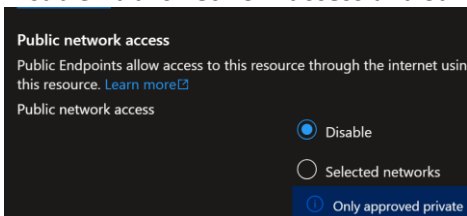
1. Go to your Azure SQL DB
2. Select your **SQL server** in the Overview



3. In your SQL Server blade, go to **Microsoft Entra ID** in *Settings* section
4. Enable **Microsoft Entra authentication only** option and **Save**

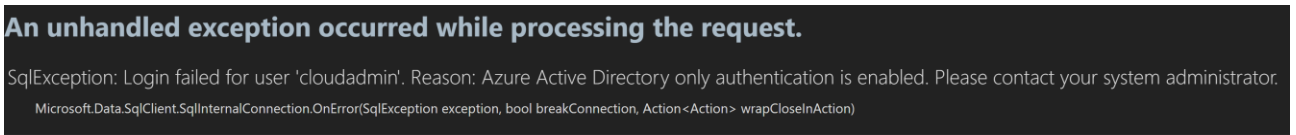


5. Go to **Networking** in *Security* section
6. Disable **Public network access** and **Save**

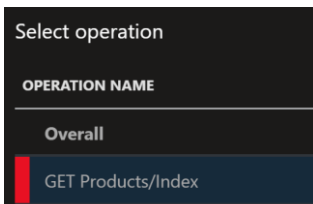


Module 3 – Test your App and Verify

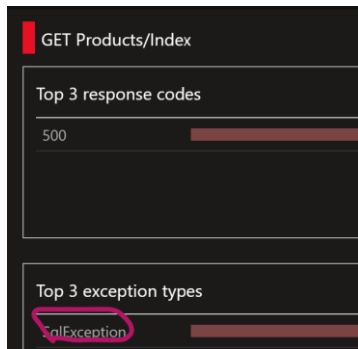
1. In VS, run you app locally by pressing F5
2. Navigate to **/products** page where the page needs connect Azure SQL DB to retrieve data
3. You will see the error **"SqlException: Login failed for user... Reason: Azure Active Directory only authentication is enabled..."**



4. Go back to your **App Insight** in Azure portal and select **Failures**
5. You will now see the new error added to your *operations* (including Top 3 exception types)



6. Select the operation that causes the application error
7. We need to check what sort of **"SqlException"** is thrown by .NET application. Select **SqlException** in *Top 3 exception types* on the right pane



8. Select the most recently recorded error details (i.e. the item might be displayed in *Suggested* section or you need to show all the errors and find the recent error)
9. The main pane shows you the detailed drill-down of all events for the call. When you select an item, you can see further details from failed method name, exception type to RequestPath and the actual error exception message
10. Read the error message details. What was the cause of the issue here?

Message	Login failed for user 'cloudadmin'. Reason: Azure Active Directory only authentication is enabled. Please contact your system administrator.
Exception type	Microsoft.Data.SqlClient.SqlException

11. SQL Authentication expects you to use only Entra ID Authentication (Azure Active Directory). Remember our connection string? We're using SQL Authentication method instead of Entra ID Authentication
12. Fix the error (i.e. disable Azure Active Directory only authentication method for your SQL Server)
13. In VS, run the app locally and navigate to */products* page
14. What do you see now?
15. Another error!

Module 4 – Troubleshoot Application Error

1. This time, you get another type of SQL exception – “Connection was denied since Deny Public Network Access is set to Yes...”

An unhandled exception occurred while processing the request.

SqlException: Reason: An instance-specific error occurred while establishing a connection to SQL Server. Connection was denied since Deny Public Network Access is set to Yes (<https://docs.microsoft.com/azure/azure-sql/database/connectivity-settings#deny-public-network-access>). To connect to this server, use the Private Endpoint from inside your virtual network (<https://docs.microsoft.com/azure/sql-database/sql-database-private-endpoint-overview#how-to-set-up-private-link-for-azure-sql-database>).

2. We don't need to go back to App Insight and find out the cause of the error, as you can clearly see what is wrong on the UI
3. Fix the error (i.e. Allow public network access and check your IP address is added to the safe list)
4. Run the app locally and navigate to */products* page
5. It's working again!

IMPORTANT

The goal of error handling in your application is not just to catch bugs - it's to ensure that when something goes wrong, your users can **continue using the app without confusion, frustration, or data loss**.

Unhandled exceptions are dangerous because they **expose internal details**—like stack traces or database paths—directly to the user. This not only breaks the user experience but can also pose security risks.

A well-designed application should **fail gracefully** - show a friendly message, log the technical details for developers, and allow the user to recover or continue without disruption.

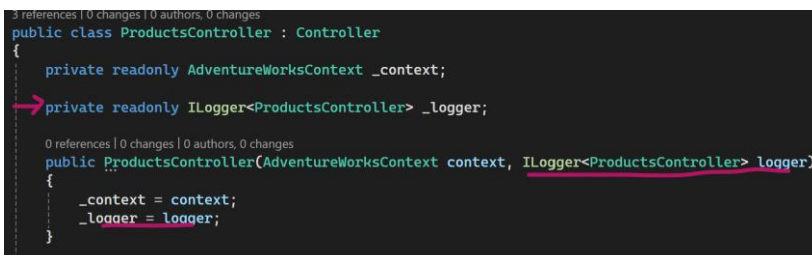
Module 5 – Implement Exception Handling

1. So far, we have seen the **unhandled** error messages directly on the page when we tried to navigate to the *Products* page due to SQL errors. But we could also see a full **stack trace** that exposes code segment, details, used technologies etc...
2. This is a bad practice. You need to mitigate this risk and make your application well-designed to handle exceptions more efficiently and gracefully
3. In C#.NET, you use **try-catch** statement to catch an exception(s) and handle errors in code. Let's implement some error handling in your code
4. In VS, open **Controllers -> ProductsController.cs**
5. Remember the SQL errors? How were those errors created in the first place?
6. When you try to navigate to the *Products* page that tries to connect to the backend SQL Database to load the product data. You will implement exception handling to catch any SQL errors in code
7. Add *ILogger* reference in the class. So, you can start logging (e.g. into App Insights, or any other logging tool(s))

```
private readonly ILogger<ProductsController> _logger;
```

8. Update the class Constructor to accept the Logger parameter to be passed over

```
public ProductsController(AdventureWorksContext context, ILogger<ProductsController> logger)
{
    _context = context;
    _logger = logger;
}
```



```
3 references | 0 changes | 0 authors, 0 changes
public class ProductsController : Controller
{
    private readonly AdventureWorksContext _context;
    private readonly ILogger<ProductsController> _logger;

    0 references | 0 changes | 0 authors, 0 changes
    public ProductsController(AdventureWorksContext context, ILogger<ProductsController> logger)
    {
        _context = context;
        _logger = logger;
    }
}
```

9. Find *Index()* method and update the code by adding **"try-catch"** statement. Paste this code at the beginning of the method first, and then replace **"// <Replace...>"** comment with the existing function code

```
try
{
    // <<Replace this with the existing code block>>
}
catch (Exception ex)
{
    // Log the exception using the injected logger
    _logger.LogError(ex, "An error occurred while retrieving products.");

    // Log the exception (logging not shown here)
    ViewData["ErrorMessage"] = "An error occurred while retrieving products.";
    return View("Error");
}
```

}

```
// GET: Products
3 references | 0 changes | 0 authors, 0 changes
public async Task<ActionResult> Index(string searchString)
{
    try
    {
        var products = from p in _context.Products
                        select p;

        if (!string.IsNullOrEmpty(searchString))
        {
            products = products.Where(p => p.Name.Contains(searchString));
        }

        // Store the current filter in ViewData so it can be reused in the view
        ViewData["CurrentFilter"] = searchString;

        return View(await products.ToListAsync());
    }
    catch (Exception ex)
    {
        // Log the exception using the injected logger
        _logger.LogError(ex, "An error occurred while retrieving products.");

        // Log the exception (logging not shown here)
        ViewData["ErrorMessage"] = "An error occurred while retrieving products.";
        return View("Error");
    }
}
```

10. **Save** changes and **rebuild** the application to ensure that you have no errors in code



IMPORTANT – Exception Handling

Exception/error handling is also about **anticipating where things can go wrong** and making sure your app can recover or fail gracefully.

You should **always include** exception handling in **areas where your code interacts with anything outside** its direct control. These are the most common and critical places to watch:

- **External Dependencies**
 - Connecting to databases (e.g., SQL Server, PostgreSQL)
 - Calling APIs or external services
 - Reading from or writing to message queues
- **File and Memory Operations**
 - Reading/writing files
 - Accessing streams or buffers
 - Working with unmanaged resource
- **User Input and Data Parsing**
 - Converting strings to numbers or dates
 - Deserializing JSON or XML
 - Validating form inputs

Module 6 – Run and Test Exception Handling

1. In VS, open *appsettings.json*
2. Change the existing SQL ConnectionString with some random username to generate another SQL error (e.g. "cloudadmin123")
3. At the moment, the database tool, **Entity Framework**, we used to scaffold the database context in code adds the default *ConnectionString* in *DbContext* class. We need to remove that and make it read from the app settings instead
4. Open **Models -> AdventureWorksContext.cs**

5. **OnConfiguring()** method, you can already see the warning statement

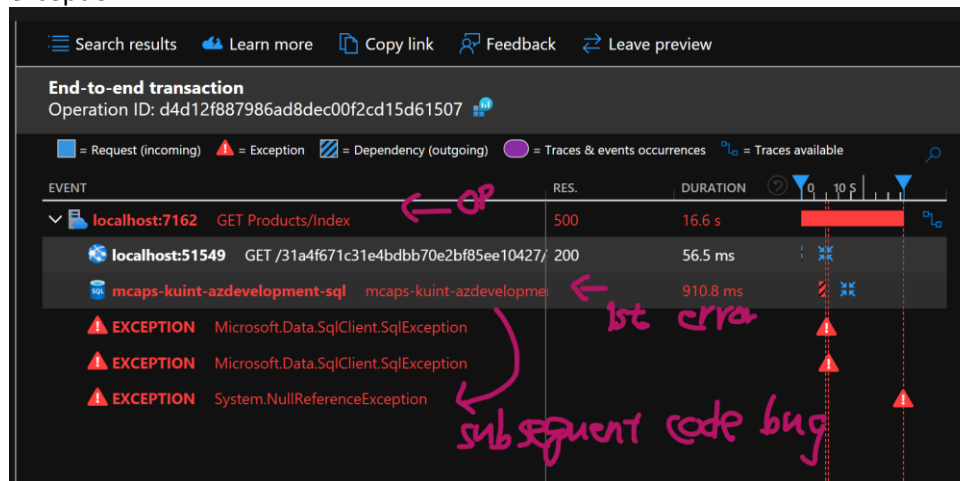
```
0 references | 0 changes | 0 authors, 0 changes
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
#warning To protect potentially sensitive information in your connection string, you should move it out of source
=> optionsBuilder.UseSqlServer("Server=tcp:microsoft-azdevelopment-sql.database.windows.net,1433;Initial
```

6. Remove that method, **OnConfiguring()**, from **AdventureWorksContext.cs**
7. If you scaffold your **DbContext** again (e.g. updating your DB context), this method will be regenerated. To avoid this, used the **-no-onconfiguring** flag when scaffolding.

e.g.

```
dotnet ef dbcontext scaffold "Name=DefaultConnection"
Microsoft.EntityFrameworkCore.SqlServer --no-onconfiguring
```

8. Run the app locally by pressing F5
9. Navigate to **/products** page
10. You get another error!
11. The is relating to an unhandled error on **null reference "@if (Model.ShowRequestId)"** in **Error.schtml** when the error page is trying display the page content on a specific condition (when **ShowRequestId** is not null)
12. When you look at the error trace in App Insight, the actual error was caused by the underlying SQL exception



13. Go back to your App Insight and find the most recent failure and check the details like above
14. Fix the SQL Connection String error for now. Don't worry about fixing code bug in **Error.cshtml**

IMPORTANT – Unit Testing “Code that works in Practice!”

Writing code that compiles is just the beginning. Writing code that works **reliably** is the goal. That's where **unit testing** comes in.

- Write tests as you code, not after
- Always write tests when changing existing functionality to avoid regressions
- Test edge cases and failure scenarios, not just the happy path
- Use mocks or fakes for external dependencies (like databases or APIs)

Think of unit tests as your safety net. They give you the confidence to refactor, improve, and scale your code without fear of breaking things.

Module 7 (Optional) – Write Unit Testing

Step 1 – Create a Unit Test Project in VS

1. In VS2022, right-click your solution and select **Add -> New Project**
2. Choose **xUnit Test Project (.NET 8)** -> Name it **MyApp.Tests** (or something like "AdventureWorksMvsApp.Tests")
3. Click **Next** and select **.NET 8** and **Create**

Step 2 – Add Project Reference

1. Right-click *Dependencies* in MyApp.Tests -> **Add Project Reference...**
2. Select your *main MVC project* (e.g., MyApp.Web) -> Click **OK**

Step 3 – Add NuGet Packages

3. Install the following packages in the test project:
 - a. Microsoft.Data.SqlClient
 - b. Microsoft.Extensions.Configuration
 - c. Microsoft.Extensions.Configuration.Json
 - d. Microsoft.Extensions.Configuration.EnvironmentVariables
 - e. Microsoft.ApplicationInsights
4. You can do this via NuGet Package Manager or the Package Manager Console:

```
Install-Package Microsoft.Data.SqlClient
Install-Package Microsoft.Extensions.Configuration
Install-Package Microsoft.Extensions.Configuration.Json
Install-Package Microsoft.Extensions.Configuration.EnvironmentVariables
Install-Package Microsoft.ApplicationInsights
```

Step 4 – Add appsettings.json to the Test Project

1. Right-click the test project -> **Add -> New Item -> JSON File** -> Name it **appsettings.json**
2. Add your connection string:

```
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=tcp:your-server.database.windows.net,1433;Initial Catalog=your-db;User ID=your-user;Password=your-password;Encrypt=True;"
  }
}
```

Step 5 – Write Unit Test

1. Rename the file, *UnitTest1.cs*, to **DatabaseConnectivityTests.cs**
2. Insert the following code block and replace the existing one

```
using System;
using Microsoft.Data.SqlClient;
using Microsoft.Extensions.Configuration;

namespace AdventureWorksMvsApp.Tests
{
    public class DatabaseConnectivityTests
    {
        private readonly string _connectionString;
```



```
public DatabaseConnectivityTests()
{
    var config = new ConfigurationBuilder()
        .SetBasePath(AppContext.BaseDirectory)
        .AddJsonFile("appsettings.json")
        .AddEnvironmentVariables()
        .Build();

    _connectionString = config.GetConnectionString("DefaultConnection")
        ?? throw new InvalidOperationException("Connection string 'DefaultConnection' not found.");
}

[Fact]
public void CanConnectToAzureSqlDatabase()
{
    using var connection = new SqlConnection(_connectionString);

    try
    {
        connection.Open();
        Assert.True(connection.State == System.Data.ConnectionState.Open);
    }
    catch (Exception ex)
    {
        Assert.Fail($"Failed to connect to DB: {ex.Message}");

        // Optionally log to Application Insights here
    }
}
}
```

Step 6 – Build the Solution

1. Press **Ctrl + Shift + B** or go to **Build > Build Solution**
2. Make sure there are no build errors

Step 7 – Open the Test Explorer

1. Go to **Test > Test Explorer** from the *top menu*
2. If it's not visible, you can also open it via **View > Test Explorer**
3. Click **Run All** or *right-click the test* and choose **Run Selected Tests**

Step 8 – View Results

1. The test result will show as **Passed** (green check) or **Failed** (red X).
2. If it fails, click the test to view the **error message and stack trace**

Summary



ACHIEVEMENTS



After you have completed the exercise, you are now able to:

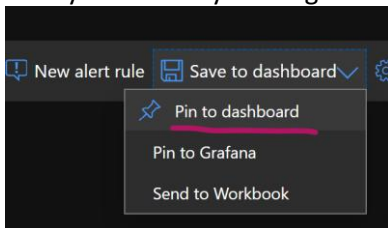
- ✓ Simulate exception/error cases
- ✓ Use App Insight to trace and track down the error cause(s)
- ✓ Learn the importance of exception/error handling for your application
- ✓ Learn the importance of unit testing
- ✓ Learn how to write unit testing

Exercise 3- Explore Metric and Availability (20min)

Topics	In this exercise, we will cover the following topics. ➤ Create Metrics Chart
Duration	• 20 Min
Tool(s)	• Azure portal
Subscription	[selected subscription]
Resource Group	[selected RG]
Reference(s)	<ul style="list-style-type: none"> • Metrics in Application Insights - Azure Monitor - Azure Monitor Microsoft Learn • Application Insights availability tests - Azure Monitor Microsoft Learn

Module 1 – Create Metrics Chart(s)

- In Azure Portal > Application Insights > **Metrics** in *Monitoring* section:
 - Add a chart  for **Server response time**
 - Add another *metric* for **Failed requests** in the *same* chart
- Rename  the chart you've created to **"AdventureWorks web app (.NET+SQL) Health Chart"**
- Save your chart by clicking the **"Pin to dashboard"** button at the top



- You can save your chart to the existing Dashboard or a new one (e.g. AppInsights Dashboard)



Application Insights –Metrics

[Metrics in Application Insights - Azure Monitor - Azure Monitor | Microsoft Learn](#)

Application Insights supports three different types of metrics: **standard (preaggregated)**, **log-based**, and **custom metrics**.

Each one brings a unique value in monitoring application health, diagnostics, and analytics.

Module 2 – Add Availability Test

- Set up an Availability Test to monitor whether your web app is reachable and responding correctly from different locations. A **URL Ping Test** is a simple test that sends HTTP requests to your app's URL
- Go to **Availability** in *Investigate* section
- +Add Classic Test** – Standard test or Classic test (simpler for beginners)
 - Test name:** *HomepageAvailabilityTest*
 - Type:** URL ping test
 - URL:** (your app's public endpoint, e.g. <https://yourapp.azurewebsites.net>)
 - Frequency:** 5 minutes
 - Locations:** Choose *2 locations*

4. Optional Settings

a. Success criteria

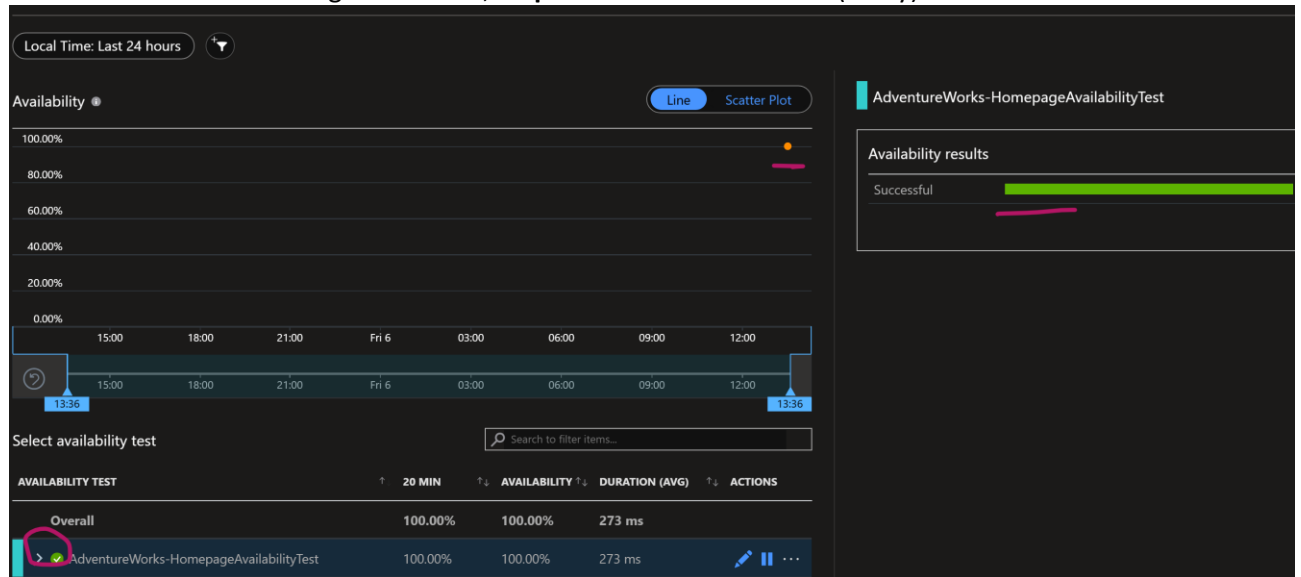
- i. HTTP response code: 200
- ii. Timeout: 30 seconds

b. Alerts: (leave unchecked)

5. Click **Create** to save the test

Module 3 – View the Availability Test Results

1. After few minutes, return to the **Availability** blade
2. You will see a chart showing **test results, response times and failures** (if any)



Application Insights –Availability

[Application Insights availability tests - Azure Monitor | Microsoft Learn](#)

Application Insights enables you to set up *recurring web tests* that monitor your website or application's availability and responsiveness from various points around the world.

These availability tests send web requests to your application at regular intervals and alert you if your application isn't responding or if the response time is too slow.

- Don't require any modifications to the website or application you're testing
- Work for any HTTP or HTTPS endpoint accessible from the public internet, including REST APIs

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Create your own Metrics chart(s)
- ✓ Add the chart(s) to Dashboard
- ✓ Create Availability Test to monitor application health and uptime
- ✓ View Availability test results

Exercise 4- Configure Diagnostic Settings and Alerts (20min)

Topics	In this exercise, we will cover the following topics. ➤
Duration	<ul style="list-style-type: none"> 20 Min
Tool(s)	<ul style="list-style-type: none"> Azure portal
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Enable Diagnostic Settings for Telemetry Data

3. Select **Diagnostic settings** in *Monitoring* section
4. Click + **Add diagnostic setting**
5. Configure the Settings
 - a. **Name:** SendToLogAnalytics
 - b. Check the boxes for:
 - i. *Request*
 - ii. *Exception*
 - iii. *Trace*
 - iv. *Dependency*
 - c. **Destination:** (You might have the default Log Analytics Workspace created for your Cloud Shell. If you don't have your own Log Analytics Workspaces, then quickly create one in another browser tab)
 - i. Choose an existing workspace or create a new one
6. Click **Save**
7. Now your telemetry is being sent to Log Analytics for deeper analysis



Diagnostic Settings

[Diagnostic settings in Azure Monitor - Azure Monitor | Microsoft Learn](#)

Diagnostic Settings allow you to export **telemetry data** (like *logs*, *metrics*, and *traces*) from Application Insights to other Azure services such as:

- Log Analytics (for querying and analysis)
- Storage Account (for long-term archiving)
- Event Hub (for streaming to external systems)

Without Diagnostic Settings, your telemetry stays only in Application Insights.

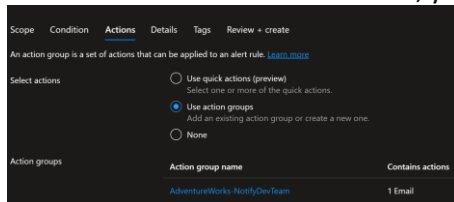
By sending it to **Log Analytics**, you can:

- Run advanced queries using Kusto Query Language (KQL)
- Correlate logs across multiple services
- Set up custom alerts and dashboards
- Retain data for longer periods

Module 2 – Create an Alert for Failed Requests

1. Select **Alerts** in *Monitoring* section -> + **Create** -> **Alert rule**

- a. **Scope:** (already set to your App Insight)
- b. **Signal name:** Failed Request
 - i. **Threshold type:** Static
 - ii. **Aggregation type:** Count
 - iii. **Value is:** Greater than
 - iv. **Unit:** Count
 - v. **Threshold:** 1
 - vi. **Check every:** 5 minutes
 - vii. **Lookback period:** 5 minutes
2. Click Next
3. Select **Use action groups**
4. **+ Create action group**
 - a. *Region:* Global
 - b. *Action group name:* NotifyDevTeam
 - c. *Display name:* Alert-AdvApp
 - d. *Notifications*
 - i. Email/SMS message/Push/Voice – enter your email address
 - ii. Name: Alerts-Adventure-Works-App
5. Click **Review+ create** to create an *Action Group*
6. Back to *Create an alert rule* wizard, your action group is added



7. Click **Net: Details**
 - a. **Severity:** 2 - Warning
 - b. **Alert rule name:** FailedRequestAlert
 - c. **Advanced options** – Enable upon creation
8. Click **Review + create** -> **Create**

Module 3 – Simulate Crash and Validate Alerts

1. Ensure the **Crash Action** exists in your application – *HomeController*
2. In VS, open **Controllers -> HomeController.cs**
3. Add this method if its not already there

```
public IActionResult Crash()
{
    throw new Exception("Simulated crash for alert testing");
}
```

4. Run your app locally
5. Navigate to **/Home/Crash** in the address bar at the top
6. Refresh the page 2~3 times within a few minutes to ensure the *“failed request count > 1”*
7. Your alert will be sent to you (after 5+min interval)

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

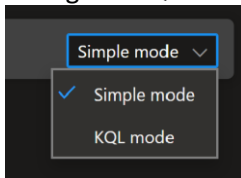
- ✓ Enable Diagnostic settings for your application
- ✓ Send logs and metrics to Log Analytics Workspace
- ✓ Create Action Group(s) for alerts
- ✓ Set alerts based on FailedRequests
- ✓ Send email alerts to yourself for failure requests

Exercise 5- View Logs and Run Kusto Queries 30(min)

Topics	In this exercise, we will cover the following topics. ➤
Duration	<ul style="list-style-type: none"> 30 Min
Tool(s)	<ul style="list-style-type: none"> Azure portal
Subscription	[selected subscription]
Resource Group	[selected RG]

Module 1 – Open Logs

1. In your App Insight in Azure, select **Logs** in *Monitoring* section. This opens the *Log Analytics workspace* where you can run KQL queries
2. Ignore and close **Queries hub window**
3. Change to KQL mode, if not done a the top right corner



4. Copy and paste each of the following queries into the query editor and click **Run**
5. **View all requests** to show all incoming HTTP requests, most recent first

```
requests
| order by timestamp desc
```

6. **Find failed requests** by filtering only the requests that failed (e.g. 500 errors)

```
requests
| where success == false
| order by timestamp desc
```

7. **View exceptions** to display all exceptions thrown by your application

```
exceptions
| order by timestamp desc
```

8. **Find slowest request** by showing the top 5 slowest request by response time

```
requests
| order by duration desc
| take 5
```

9. **Filter by URL or operation name** – useful for investigating specific endpoints

```
requests
| where url contains "/Crash"
| order by timestamp desc
```

Module 2 – Explorer and Modify Queries

1. Try changing the take value to see more results
2. Add a *where timestamp > ago(1h)* clause to filter by time
3. Use project to select specific columns:

requests

| project timestamp, name, url, duration, success

Summary



ACHIEVEMENTS

After you have completed the exercise, you are now able to:

- ✓ Access and explore telemetry using **Log Analytics**
- ✓ Write and run basic **KQL** queries
- ✓ Filter and sort data to *diagnose issues* and *monitor performance*

This is the end of the lab.

Well done!