Azure Cloud Fundamentals

Lab 2

Building and Deploying Your First .NET Web App

From empty project to running application on Azure App Service

VS Code / Cursor / Windsurf / Visual Studio → .NET 10 → Azure App Service

Setting	Value
Module	Application development and deployment
Level	Fundamentals (AZ-900 aligned)
Estimated time	90 – 150 minutes
Estimated cost	Same as Lab 1 — uses the resources you already deployed
Prerequisite	Lab 1 — "Deploying a Multi-Tier Web Application on Azure"
Deployment method	CLI build + portal / VS Code deploy

1. Lab Objective

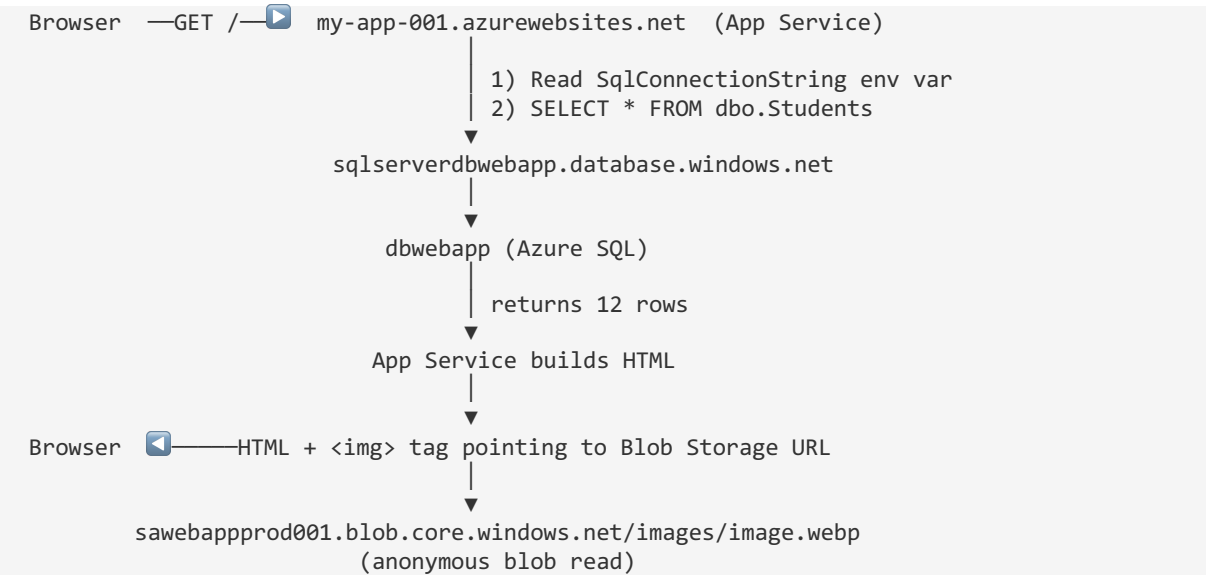
In Lab 1 you built the infrastructure — an empty web app, an empty database, and an empty blob container. In this lab you will build, deploy, and configure the actual application code that ties all three together.

By the end of this lab you will have:

- Installed the .NET 10 SDK and an IDE of your choice
- Created a new ASP.NET Core minimal web app from scratch using `dotnet new web`
- Added the `Microsoft.Data.SqlClient` NuGet package to connect to Azure SQL
- Replaced the default Program.cs with a working app that queries the Students table and displays it with an image from Blob Storage
- Built, published, and packaged the app as a ZIP
- Deployed the ZIP to App Service using either the portal or the VS Code Azure extension
- Configured the App Service with the SQL connection string and the correct startup command
- Seen your application render a live HTML page with data from Azure SQL and an image from Azure Blob Storage

Architecture recap — what the app does

When a user opens the web app URL, the following sequence happens:



> **NOTE**
> The HTML page is built entirely on the server (a single GET / endpoint). The browser only makes one additional request — directly to Blob Storage — to fetch the image. There is no JavaScript, no REST API, no frontend framework. This is the simplest possible shape of a three-tier web application.

2. Prerequisites

Before starting this lab, confirm the following:

From Lab 1

- The resource group `rg-web-apps-001` exists with all five resources (App Service Plan, Web App, SQL Server, SQL Database, Storage Account)
- The SQL Database `dbwebapp` has the `dbo.Students` table populated with 12 rows
- The blob container `images` exists with **Blob** anonymous access level
- You remember the SQL admin password for user `sqldba`

Local development environment

- **.NET 10 SDK** installed (see Section 3 for instructions)
- **One of these IDEs:** Visual Studio Code, Cursor, Windsurf, or Visual Studio 2022+ (see Section 4)
- **A sample image file** in `.webp` format (see Section 5 — any small image will do; you can convert a PNG or JPG online)

Optional but helpful

- **Azure CLI** if you want to use the `az webapp deploy` approach (see Appendix C)
- **Git** if you want to version-control your project

> ⚠ **WARNING**
> Because Azure Storage and SQL Server names are globally unique, each student has a slightly different resource set. Two places in this lab require you to substitute YOUR own names: (1) the image URL in Program.cs, and (2) the SQL connection string. Both spots are clearly marked in the steps.

3. Install the .NET 10 SDK

Your App Service in Lab 1 was configured with Runtime Stack .NET Core 10.0, so your local SDK must match. Using a different major version will cause the app to fail with a runtime mismatch error.

Windows

1. Open `https://dotnet.microsoft.com/download` in your browser.
2. Download the .NET 10 SDK installer for Windows x64.
3. Run the installer, accept the defaults, and finish.

macOS

1. Install via Homebrew:

```
brew install --cask dotnet-sdk
```

2. Or download the pkg installer from `https://dotnet.microsoft.com/download`.

Linux (Ubuntu / Debian)

1. Install via apt:

```
sudo apt update
sudo apt install -y dotnet-sdk-10.0
```

2. Or use the official install script from `https://dotnet.microsoft.com/download`.

Verify the installation

Open a new terminal (or PowerShell on Windows) and run:

```
dotnet --version
```

You should see a version starting with `10.` (for example `10.0.100`). If you get a "command not found" error, close and reopen your terminal so the new PATH takes effect, or restart your machine.

> 🚧 **COMMON PITFALL**
>
> If you see a version like `8.0.x` or `9.0.x`, you have an older SDK. You can have multiple .NET SDKs installed side by side — newer SDK becomes the default for `dotnet new` and `dotnet build`. To confirm which SDKs are installed, run `dotnet --list-sdks`.

4. Choose and Configure Your IDE

You can use any of the following. Pick one and follow only its sub-section.

4.1 Visual Studio Code, Cursor, or Windsurf

Cursor and Windsurf are forks of Visual Studio Code, so they share the same extensions and the same workflow. The instructions below apply to all three — wherever it says "VS Code", substitute your IDE.

1. Install your editor if you have not already:
   - VS Code: `https://code.visualstudio.com`
   - Cursor: `https://cursor.com`
   - Windsurf: `https://windsurf.com`
2. Open the editor and go to the **Extensions** panel (Ctrl+Shift+X on Windows/Linux, Cmd+Shift+X on Mac).
3. Search for and install the following extensions:
   - **C# Dev Kit** (publisher: Microsoft) — adds C# language support, IntelliSense, and debugging
   - **Azure Tools** (publisher: Microsoft) — lets you deploy directly to App Service from the editor
4. Sign in to Azure from the editor:
   - Open the Command Palette (Ctrl+Shift+P / Cmd+Shift+P) and run `Azure: Sign In`.
   - Complete the browser-based authentication flow.
   - After signing in, open the Azure panel from the left sidebar and confirm you can see your subscription and the `rg-web-apps-001` resource group.

4.2 Visual Studio 2022 or 2026 (Windows only)

1. Install Visual Studio from `https://visualstudio.microsoft.com` (Community edition is free for individuals and students).
2. During installation, make sure the following workloads are selected:
   - **ASP.NET and web development**
   - **Azure development**
   - **.NET desktop development** (optional but helpful)
3. After installation, sign in to Visual Studio with the Microsoft account that has access to your Azure subscription.

> 💡 **TIP**
> Cursor and Windsurf both support AI-assisted development out of the box, which can speed up exploration of Azure SDK APIs. However, always read the generated code carefully — AI assistants occasionally use outdated `System.Data.SqlClient` (legacy) instead of `Microsoft.Data.SqlClient` (current). This lab uses the current one.

5. Upload a Test Image to Blob Storage

The application code references a blob at `https://<your-storage>.blob.core.windows.net/images/image.webp`. You need to upload a file with exactly that name before the app will show an image.

1. Find or prepare any small image (under 1 MB is plenty). Convert it to `image.webp` using any online converter (for example `https://cloudconvert.com/png-to-webp`). The filename MUST be exactly `image.webp`.
2. In the Azure Portal, navigate to your storage account (e.g. `sawebappprod001`).
3. In the left menu, click **Data storage** → **Containers**, then open the `images` container.
4. Click **Upload** at the top.
5. Browse to your `image.webp` file, select it, and click **Upload**.
6. Click the uploaded blob and copy the **URL** shown in the Overview pane. It should look like:

```
https://sawebappprod001.blob.core.windows.net/images/image.webp
```

7. Open that URL in a new incognito browser tab. The image should load with no authentication prompt (because the container is set to Blob-level anonymous access).

> ⚠ **WARNING**
>
> **Write down your exact blob URL now.** If your storage account name is NOT `sawebappprod001`, you will need to replace that name in Program.cs in Section 9.

6. Create the .NET Project

We will use the .NET CLI to scaffold a minimal ASP.NET Core web app. This creates a standard project structure that works identically on Windows, macOS, and Linux.

1. Open a terminal in a folder where you want your project to live. For example:

```
# Windows (PowerShell)
cd $env:USERPROFILE\source\repos

# macOS / Linux
cd ~/source/repos
```

2. Create a new empty web project:

```
dotnet new web -n StudentDemoApp
```

This creates a `StudentDemoApp` folder containing a minimal ASP.NET Core web app: a `Program.cs`, a project file (`StudentDemoApp.csproj`), and some default settings.

3. Change into the project directory:

```
cd StudentDemoApp
```

4. Add the Microsoft.Data.SqlClient NuGet package so you can connect to Azure SQL:

```
dotnet add package Microsoft.Data.SqlClient
```

This modifies `StudentDemoApp.csproj` to add a `<PackageReference>` and downloads the package into your local NuGet cache.

5. Open the project in your IDE:

```
# VS Code / Cursor / Windsurf
code .            # or: cursor .   or: windsurf .

# Visual Studio (Windows)
# Double-click StudentDemoApp.csproj
```

6. Verify your project structure looks like this:

```
StudentDemoApp/
├── Properties/
│   └── launchSettings.json
├── appsettings.json
├── appsettings.Development.json
├── Program.cs
└── StudentDemoApp.csproj
```

> **NOTE**
>
> The `dotnet new web` template uses the **minimal hosting API** (introduced in .NET 6). There is no `Startup.cs` file — the whole app configuration and routing lives in `Program.cs`. This is modern ASP.NET Core.

7. Replace Program.cs With the Lab Application

Now you will replace the generated default Program.cs with a complete working application that connects to SQL and renders an HTML page.

1. In your IDE, open `Program.cs` in the project root.
2. Select all existing content (Ctrl+A / Cmd+A) and delete it.
3. Copy the full Program.cs source from **Appendix A** at the end of this document and paste it into the empty file.
4. Save the file (Ctrl+S / Cmd+S).
5. Build the project to confirm it compiles:

```
dotnet build
```

6. You should see `Build succeeded` at the bottom. If you see errors, make sure you pasted the code correctly and that `Microsoft.Data.SqlClient` was installed in Section 6.

Walkthrough: what the code actually does

Before you run it, take a minute to understand the structure. The code is about 130 lines and handles three scenarios:

Scenario	What the code does	What you see in the browser
No connection string	Detects the missing SqlConnectionString env var and returns a simple error HTML page	Red error: "Missing environment variable: SqlConnectionString"
Connection or query fails	Catches the SqlException (or any exception) and returns the exception message HTML-encoded	Red error: "Database error: [exception message]"
Everything works	Executes SELECT on dbo.Students, builds a full HTML page with an <img> tag and a table of all students	A styled card with the image and a table of 12 students

Key parts of the code

- `WebApplication.CreateBuilder(args)` and `app.MapGet("/", ...)` — the minimal hosting API. One endpoint, GET on the root path.
- `Environment.GetEnvironmentVariable("SqlConnectionString")` — reads the connection string from the runtime environment. On Azure, this comes from the App Service environment variables you will configure in Section 14.
- The SQL query only returns 8 specific columns (`StudentId, FirstName, LastName, Email, CourseName, YearLevel, GPA, IsActive`) — **not** SELECT *. This is a good production habit: return only what the UI needs.
- The `StudentRow` class at the bottom is a simple DTO (Data Transfer Object) to hold each row as a strongly-typed object before converting it to HTML.

- `WebUtility.HtmlEncode()` — encodes user-supplied data before writing it into the HTML output. This prevents HTML injection if, for example, a student's name contained `<script>` tags.
- The inline CSS and HTML via `StringBuilder` is deliberately unsophisticated — real apps use Razor Pages, Blazor, or a separate frontend. Keeping everything in one file makes this lab easier to reason about.

8. Customize the Code for Your Environment

Because storage account and SQL server names are globally unique, the code from Appendix A is hardcoded with the instructor's names. You MUST change two things before deploying:

Change 1: the blob image URL in Program.cs

Find this line in Program.cs (around line 90):

```
<img src='https://sawebappprod001.blob.core.windows.net/images/image.webp' alt='Demo
image' width='700' />
```

Replace `sawebappprod001` with the name of YOUR storage account from Lab 1. For example, if your storage account is `sawebappim001`, the line becomes:

```
<img src='https://sawebappim001.blob.core.windows.net/images/image.webp' alt='Demo image'
width='700' />
```

> ⚠ **WARNING**
> If you skip this step, the HTML page will render but the image will show as a broken icon — the browser will try to fetch from the instructor's storage account, which you cannot read from. The text and data will still work.

Change 2: the SQL connection string (done later)

The connection string is **not** in the code. It is read from an environment variable at runtime. You will configure it in Section 12 (for local testing) and Section 14 (for Azure). The server name in that string must match **your** SQL server, not the instructor's.

> 💡 **TIP**
> If you want to keep the code 100% portable and never hardcode the storage account, a better pattern is to read the blob URL from configuration: pass it in as an environment variable like BlobImageUrl. For this first lab we keep it simple so you can see exactly where the URL lives.

9. Test Locally (Optional but Recommended)

Before deploying to Azure, it is a good idea to run the app on your local machine and confirm it can reach your Azure SQL Database. This catches problems (bad password, firewall, wrong connection string) in a much faster feedback loop than re-deploying to Azure each time.

9.1 Build your connection string

Build the connection string using this template — replace the four <placeholder> values with yours:

```
Server=tcp:<YOUR-SQL-SERVER>.database.windows.net,1433;Initial Catalog=dbwebapp;Persist Security Info=False;User ID=sqldba;Password=<YOUR-PASSWORD>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
```

Replace:

- `<YOUR-SQL-SERVER>` → your SQL server name (e.g. `sqlserverdbwebapp` — the part BEFORE `.database.windows.net`)
- `<YOUR-PASSWORD>` → the SQL admin password you set for user `sqldba` in Lab 1

> 💡 **TIP**
> You can also copy a pre-filled template from the portal: navigate to your SQL database `dbwebapp` → **Show database connection strings** (top right of the Overview blade) → copy the **ADO.NET (SQL authentication)** string. You still need to replace `{your_password}` with your real password.

9.2 Set the environment variable for your local session

Windows (PowerShell)

```
$env:SqlConnectionString = "Server=tcp:sqlserverdbwebapp.database.windows.net,1433;Initial Catalog=dbwebapp;User ID=sqldba;Password=YourPasswordHere;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
```

macOS / Linux (bash or zsh)

```
export SqlConnectionString="Server=tcp:sqlserverdbwebapp.database.windows.net,1433;Initial Catalog=dbwebapp;User ID=sqldba;Password=YourPasswordHere;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;"
```

> ⚠️ **WARNING**
> This sets the variable ONLY in the current terminal session. If you open a new terminal, you need to set it again. Never commit the password to Git.

9.3 Make sure your local IP is in the SQL firewall

In Lab 1 you added a firewall rule for your public IP at the time. If your IP has changed, you will get a SQL exception when running the app. Fix: go to the SQL server `sqlserverdbwebapp` → **Networking** → click **Add your client IPv4 address** again, then click **Save**.

9.4 Run the app

In the same terminal (where you set the env var), from the StudentDemoApp folder:

```
dotnet run
```

You should see output like:

```
Building...
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
```

Open `http://localhost:5000` in your browser. You should see the title, the image, and the table of students. Press `Ctrl+C` in the terminal to stop the app.

> 🚧 **COMMON PITFALL**
>
> **Database error: Cannot open server ... requested by the login.** Means your public IP is not in the SQL firewall. Add it as described in step 9.3.

> 🚧 **COMMON PITFALL**
>
> **Database error: Login failed for user 'sqldba'.** Means the password in the connection string is wrong. Double-check for typos and avoid copy-pasting from rich-text sources that might convert quotes.

> 🚧 **COMMON PITFALL**
>
> **Database error: A connection was successfully established but then an error occurred ... certificate.** The `Encrypt=True;TrustServerCertificate=False` settings are correct for Azure SQL. If you see certificate errors locally, make sure your system clock is correct.

10. Publish the Application

Publishing compiles the app into a self-contained folder of DLLs, JSON config files, and dependencies that App Service knows how to run. This is **different** from the source code — you cannot just upload `Program.cs` and expect it to work.

1. In the StudentDemoApp folder, run:

```
dotnet publish -c Release -o ./publish
```

The `-c Release` flag produces an optimized build. The `-o ./publish` flag puts all output in a subfolder called `publish` (next to your source code).

2. When it finishes, list the contents of the publish folder:

```
# Windows
dir publish

# macOS / Linux
ls publish
```

3. You should see files like:

```
StudentDemoApp.dll
StudentDemoApp.deps.json
StudentDemoApp.runtimeconfig.json
Microsoft.Data.SqlClient.dll
appsettings.json
appsettings.Development.json
web.config        (only if published for Windows)
... many more .dll files
```

> **NOTE**
> The published output is larger than you might expect (typically 50–100 MB). That is because `dotnet publish` includes every dependency required at runtime, including `Microsoft.Data.SqlClient` and its transitive dependencies. App Service does not have these pre-installed beyond the base framework.

11. Package the Deployment ZIP

App Service's ZIP deploy endpoint (`ZipDeploy`) extracts the zip contents directly into `/home/site/wwwroot`. This means the structure of your zip matters: the files must be at the **root** of the zip, **not** inside a top-level `publish/` folder.

The rule, stated plainly

	Structure inside the zip	What happens on App Service
✅ Correct	StudentDemoApp.dll, web.config, ... at root	Files land directly in /home/site/wwwroot. App starts.
❌ Wrong	publish/StudentDemoApp.dll, publish/web.config, ...	Files land in /home/site/wwwroot/publish/. App does NOT start — the default placeholder page appears instead.

11.1 Create the zip the right way

macOS / Linux

From your project folder (the one containing the publish folder):

```
cd publish
zip -r ../StudentDemoApp.zip .
cd ..
```

The `cd publish` step is critical — you zip FROM inside the publish folder so the zip contents sit at the root. The `.` at the end of the zip command means "everything in the current directory".

> 💡 **TIP**
> If `zip` is not installed on your system, on macOS you can use `ditto` instead: `cd publish && ditto -c -k --sequesterRsrc . ../StudentDemoApp.zip`.

Windows (PowerShell)

```
Compress-Archive -Path .\publish\* -DestinationPath .\StudentDemoApp.zip -Force
```

The `\*` wildcard is important — it selects the **contents** of the publish folder, not the folder itself.

11.2 Verify the zip structure

Always verify before uploading. Open your StudentDemoApp.zip in any archive tool or run:

```
# macOS / Linux
unzip -l StudentDemoApp.zip | head -20

# Windows (PowerShell)
Get-ChildItem -Path .\StudentDemoApp.zip | Expand-Archive -DestinationPath .\verify -Force
dir .\verify
```

The first few entries in the zip should be files directly at the root:

```
StudentDemoApp.dll
StudentDemoApp.deps.json
StudentDemoApp.runtimeconfig.json
appsettings.json
...
```

If you see entries like this, STOP and re-zip correctly:

```
publish/
publish/StudentDemoApp.dll
publish/web.config
...
```

> 🚧 **COMMON PITFALL**
>
> **The "extra folder level" trap.** Every year, students spend 30+ minutes staring at the default App Service welcome page on their deployed URL. In almost every case the cause is the same: the zip has an extra `publish/` folder at the top level. Always verify the zip BEFORE uploading.

12. Deploy to Azure App Service

You have two options. Option A (ZIP deploy via portal) is reliable, explicit, and works from any OS. Option B (VS Code Azure Tools) is faster for iterative development. Pick one — you do not need to do both.

12.A Option A — ZIP Deploy via the Azure Portal

This method uses the Kudu deployment service, which is the engine behind App Service deployments. You upload the zip to the `ZipDeploy` endpoint and Kudu extracts it to `/home/site/wwwroot`.

1. In the Azure Portal, navigate to your Web App `my-app-001`.
2. In the left menu, expand **Development Tools** → click **Advanced Tools**, then click the **Go** → button. This opens Kudu in a new tab.
3. In the Kudu top menu, click **Tools** → **Zip Push Deploy**. (The URL is `https://my-app-001.scm.azurewebsites.net/ZipDeployUI`.)
4. Drag your `StudentDemoApp.zip` into the drop zone.
5. Wait for the deployment to complete. You will see progress messages and finally "Deployment successful".
6. Back in Kudu, click **Debug console** → **Bash** (or CMD).
7. Navigate to `/home/site/wwwroot` and run `ls` (or `dir`). Confirm that the files sit directly in wwwroot — **NOT** inside a publish subfolder.

> 💡 **TIP**
> If you prefer the command line, you can use Azure CLI for the same effect: `az webapp deploy --resource-group rg-web-apps-001 --name my-app-001 --src-path ./StudentDemoApp.zip --type zip`

12.B Option B — Deploy from VS Code / Cursor / Windsurf

If you installed the Azure Tools extension in Section 4, you can deploy with a few clicks without manually zipping.

1. In VS Code, open the **Azure** panel from the left sidebar.
2. Expand **App Services** → your subscription → find `my-app-001`.
3. Right-click on `my-app-001` → **Deploy to Web App...**.
4. Select your project folder `StudentDemoApp` (not the publish folder — the extension will run `dotnet publish` for you).
5. When prompted "Are you sure you want to deploy to...?", click Deploy.
6. Watch the Output panel. Deployment typically takes 1–3 minutes.
7. When complete, click Browse Website in the notification that appears.

> **NOTE**
> The Azure Tools extension uses the same `ZipDeploy` endpoint under the hood, but it handles the build, publish, and zip steps for you in a single operation. It also watches for structure mistakes and warns if it detects them.

13. Configure the App Service

After deploying the code, you still need to configure two things before the app can run: the SQL connection string (as an environment variable) and the startup command (so Linux App Service knows which DLL to launch).

13.1 Add the SqlConnectionString environment variable

1. In the portal, navigate to your Web App `my-app-001`.
2. In the left menu, click **Settings** → **Environment variables**.
3. On the **App settings** tab, click **+ Add**.
4. Enter:
   - **Name:** `SqlConnectionString` (exactly this, case-sensitive — it must match the string in Program.cs)
   - **Value:** your full connection string from Section 9.1, including your password
   - **Deployment slot setting:** leave unchecked
5. Click **Apply** to save the new setting.
6. Confirm the **Restart the app** prompt. The app will restart within a few seconds.

> ⚠️ **WARNING**
> **Use App settings, NOT Connection strings.** App Service has a separate "Connection strings" section that prefixes the name with `SQLAZURECONNSTR_` when surfaced as an env var. Because our code calls `Environment.GetEnvironmentVariable("SqlConnectionString")` directly, we must use the App settings section so the name comes through unchanged.

13.2 Set the Startup Command (Linux only)

On Linux App Service, .NET apps need to be told which assembly to run. Without an explicit startup command you will see the default "Your App Service app is up and running" placeholder page instead of your app.

1. Navigate to `my-app-001` → **Settings** → **Configuration**.
2. Click the **General settings** tab.
3. Scroll to the **Startup Command** field and enter:

```
dotnet StudentDemoApp.dll
```

4. Click **Save** at the top. Confirm the restart prompt.

> 🚧 **COMMON PITFALL**
> **The #1 cause of "why does my app still show the welcome page":** forgetting to set the Startup Command on Linux. Just changing the Stack setting (`.NET Core 10.0`) is NOT enough. You must also tell App Service how to launch your compiled assembly.

13.3 Restart the Web App explicitly

Saving either of the above settings triggers a restart, but if you changed both, it is safer to explicitly restart once more to ensure everything is fresh.

1. Navigate to `my-app-001` → **Overview**.
2. Click **Restart** at the top and confirm.
3. Wait 30–60 seconds for the restart to finish.

14. Verify the Deployment

1. Open `https://my-app-001.azurewebsites.net` in a new incognito browser tab.
2. You should see your application:
    - Title: **"Web App + Storage Account + Azure SQL Demo"**
    - A subtitle: "This image is loaded from Azure Blob Storage."
    - An image loaded from your storage account (if you updated the URL in Program.cs)
    - A table titled **"Students"** with 12 rows and columns: Student ID, First Name, Last Name, Email, Course, Year, GPA, Active
3. If you see a **red error message** instead, the app started but something is misconfigured. Read the error text carefully — Program.cs shows you the exact exception.
4. If you see the **default App Service landing page** ("Your App Service app is up and running"), the app is not starting. Go to Section 15 (Troubleshooting).

> 💡 **TIP**
> The very first request after deployment may take 30–60 seconds because (a) the App Service worker is cold-starting, and (b) the Serverless SQL database may be paused from Lab 1. Subsequent requests within the next hour will be fast.

15. Troubleshooting Common Issues

Symptom: Default App Service welcome page ("Your app is up and running")

Most likely cause: Startup Command not set, or zip was uploaded with a top-level `publish/` folder.

Diagnose:

- Go to Kudu (`https://my-app-001.scm.azurewebsites.net`) → Debug console → Bash.
- Run `cd /home/site/wwwroot && ls`.
- If you see `StudentDemoApp.dll` directly, re-check Section 13.2 (Startup Command).
- If you see a `publish/` subfolder instead, re-zip correctly per Section 11 and redeploy.

Symptom: Red error "Missing environment variable: SqlConnectionString"

Cause: App setting name is wrong, or App Service was not restarted after adding it.

Fix:

- In **Settings** → **Environment variables** → **App settings**, confirm the name is `SqlConnectionString` (case-sensitive, no spaces).
- Confirm you added it as an **App setting**, NOT a Connection string (the latter prefixes the name with `SQLAZURECONNSTR_`).
- Restart the app.

Symptom: Red error "Database error: Cannot open server ... requested by the login"

Cause: Azure SQL firewall is blocking the App Service outbound IPs.

Fix:

- Go to `sqlserverdbwebapp` → **Security** → **Networking**.
- Confirm the checkbox **Allow Azure services and resources to access this server** is checked and saved.
- Alternative (more secure): add the App Service's specific outbound IPs as firewall rules — you can find them in the Web App's **Overview** → **Outbound IP addresses**.

Symptom: Red error "Database error: Login failed for user 'sqldba'"

Cause: Wrong password in the connection string.

Fix:

- Verify the password you used in the connection string matches the SQL admin password set in Lab 1. If you don't remember, go to `sqlserverdbwebapp` → **Reset password** at the top, then update the env var.
- If the password contains special characters like `"$;`, make sure they are not being interpreted by your shell when you set the local env var. Wrap in single quotes on bash/zsh.

Symptom: Red error "Database error: Invalid object name 'dbo.Students'"

Cause: The Students table was never created. Go back to Lab 1, Section 6.7 and run the T-SQL script.

Symptom: Page loads, table shows, but image is broken

Cause: You did not update the storage account URL in Program.cs, OR you did not upload `image.webp` to the `images` container.

Fix:

- Open the image URL directly in a browser. If you get a 404 or AuthorizationFailure, the blob or container access is wrong.
- Verify the `images` container has **Anonymous access level: Blob** (not Private).
- Verify the blob filename is exactly `image.webp` (case-sensitive).
- Verify the storage account name in Program.cs matches YOUR storage account, then redeploy.

Viewing live logs

When troubleshooting, stream the live logs from App Service:

- **Portal:** `my-app-001` → **Monitoring** → **Log stream**.
- **VS Code:** right-click the Web App in the Azure panel → **Start Streaming Logs**.
- **Azure CLI:** `az webapp log tail --name my-app-001 --resource-group rg-web-apps-001`

16. Learning Points: Lab vs. Production

As in Lab 1, many of the choices here would not fly in a production environment. The table below compares.

Area	This Lab	Production Recommendation
Connection string storage	Plaintext App setting in App Service	Azure Key Vault reference in App settings, accessed via Managed Identity
SQL authentication	SQL login (sqldba) + password	Managed Identity of the App Service — no password anywhere
Hardcoded blob URL in code	sawebappprod001 baked into Program.cs	Read from IConfiguration / App settings; change per environment
Deployment method	Manual zip upload or VS Code one-click	CI/CD pipeline (GitHub Actions, Azure DevOps) with PR validation and gated deploys
Deployment target	Production slot directly	Deploy to staging slot, smoke-test, then swap
Secrets in env vars	Visible in portal to anyone with Contributor	Key Vault with RBAC restricted to specific users
Error handling	Exception message rendered into HTML	Generic error message + structured logging to Application Insights
HTML generation	StringBuilder concatenation	Razor Pages / Blazor / an API + separate frontend framework
SQL query	Inline string	Parameterized query via Dapper/EF Core; connection pooling
Logging / telemetry	None beyond App Service defaults	Application Insights wired in, with custom events and traces
Health checks	None	/health endpoint registered and configured on App Service
Scalability	Single F1 worker instance, no always-on	Autoscale rules, always-on, multiple instances across zones

Highlights worth internalizing

- **Never commit secrets to Git.** Connection strings, passwords, API keys — all of them go in Key Vault or App Settings, never in source control.
- **Prefer Managed Identity over passwords.** The App Service can authenticate to Azure SQL using its own identity, with no secret stored anywhere. This is the biggest security upgrade you can make to this lab.
- **Don't leak exceptions to users.** This lab renders the full database exception on the page. Convenient for debugging, terrible for production — it can expose server names, schema details, or internal errors to attackers.

- **Deploy via pipelines, not drag-and-drop.** A ZIP deploy from a developer laptop is unrepeatable, unaudited, and tied to one person's machine. CI/CD is the only responsible approach in shared environments.

17. Cleanup

When you are finished with both labs, clean up the same way as in Lab 1: delete the entire resource group.

1. Azure Portal → Resource groups → `rg-web-apps-001`.
2. Click **Delete resource group** at the top.
3. Type `rg-web-apps-001` to confirm, then click **Delete**.

Next steps

- Rewrite the infrastructure from Lab 1 as Bicep or Terraform, and deploy it from a GitHub Actions workflow.
- Convert the SQL authentication to Managed Identity (remove the password entirely).
- Add Application Insights and instrument the app with structured logging.
- Add a staging slot and do a blue/green swap on your next deploy.
- Migrate from inline HTML to Razor Pages for a more realistic project layout.
- Add Private Endpoints to SQL and Storage, and VNet integration to the Web App.

Appendix A — Full Program.cs Source Code

Copy the block below verbatim into your `Program.cs`. Remember to replace `sawebappprod001` in the `<img src=...>` line with your storage account name (Section 8).

```
using Microsoft.Data.SqlClient;
using System.Net;
using System.Text;

var builder = WebApplication.CreateBuilder(args);
var app = builder.Build();

app.MapGet("/", async () =>
{
    var connectionString = Environment.GetEnvironmentVariable("SqlConnectionString");

    if (string.IsNullOrWhiteSpace(connectionString))
    {
        var missingHtml = @"
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8' />
    <title>Student Demo</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 40px; }
        .error { color: darkred; font-weight: bold; }
    </style>
</head>
<body>
    <h1>Web App + Azure SQL Demo</h1>
    <p class='error'>Missing environment variable: SqlConnectionString</p>
</body>
</html>";
        return Results.Content(missingHtml, "text/html");
    }

    var students = new List<StudentRow>();

    try
    {
        await using var connection = new SqlConnection(connectionString);
        await connection.OpenAsync();

        const string sql = @"
SELECT StudentId, FirstName, LastName, Email, CourseName, YearLevel, GPA, IsActive
FROM dbo.Students
ORDER BY StudentId";

        await using var command = new SqlCommand(sql, connection);
        await using var reader = await command.ExecuteReaderAsync();

        while (await reader.ReadAsync())
        {
            students.Add(new StudentRow
            {
                StudentId = reader.GetInt32(0),
                FirstName = reader.GetString(1),
                LastName = reader.GetString(2),
```

```
                Email = reader.GetString(3),
                CourseName = reader.GetString(4),
                YearLevel = reader.GetInt32(5),
                GPA = reader.IsDBNull(6) ? null : reader.GetDecimal(6),
                IsActive = reader.GetBoolean(7)
            });
        }
    }
    catch (Exception ex)
    {
        var errorHtml = @"
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8' />
    <title>Student Demo</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 40px; }
        .error { color: darkred; font-weight: bold; }
    </style>
</head>
<body>
    <h1>Web App + Azure SQL Demo</h1>
    <p class='error'>Database error: " + WebUtility.HtmlEncode(ex.Message) + @"</p>
</body>
</html>";
        return Results.Content(errorHtml, "text/html");
    }

    var html = new StringBuilder();

    html.Append(@"
<!DOCTYPE html>
<html>
<head>
    <meta charset='utf-8' />
    <meta name='viewport' content='width=device-width, initial-scale=1' />
    <title>Student Demo</title>
    <style>
        body { font-family: Arial, sans-serif; margin: 40px; background: #f7f7f7; }
        .container { max-width: 1200px; margin: 0 auto; background: white; padding: 30px;
border-radius: 12px; }
        h1 { margin-bottom: 10px; }
        p { margin-bottom: 20px; }
        img { max-width: 100%; border-radius: 8px; margin-bottom: 30px; }
        table { border-collapse: collapse; width: 100%; margin-top: 20px; }
        th, td { border: 1px solid #ddd; padding: 10px; text-align: left; }
        th { background-color: #f4f4f4; }
    </style>
</head>
<body>
    <div class='container'>
        <h1>Web App + Storage Account + Azure SQL Demo</h1>
        <p>This image is loaded from Azure Blob Storage.</p>
        <img src='https://sawebappprod001.blob.core.windows.net/images/image.webp'
alt='Demo image' width='700' />
        <h2>Students</h2>
        <table>
            <thead>
                <tr>
                    <th>Student ID</th>
                    <th>First Name</th>
```

```
                    <th>Last Name</th>
                    <th>Email</th>
                    <th>Course</th>
                    <th>Year</th>
                    <th>GPA</th>
                    <th>Active</th>
                </tr>
            </thead>
            <tbody>
");

    foreach (var s in students)
    {
        html.Append("<tr>");
        html.Append("<td>" + s.StudentId + "</td>");
        html.Append("<td>" + WebUtility.HtmlEncode(s.FirstName) + "</td>");
        html.Append("<td>" + WebUtility.HtmlEncode(s.LastName) + "</td>");
        html.Append("<td>" + WebUtility.HtmlEncode(s.Email) + "</td>");
        html.Append("<td>" + WebUtility.HtmlEncode(s.CourseName) + "</td>");
        html.Append("<td>" + s.YearLevel + "</td>");
        html.Append("<td>" + (s.GPA.HasValue ? s.GPA.Value.ToString("0.00") : "") +
"</td>");
        html.Append("<td>" + (s.IsActive ? "Yes" : "No") + "</td>");
        html.Append("</tr>");
    }

    html.Append(@"
            </tbody>
        </table>
    </div>
</body>
</html>");

    return Results.Content(html.ToString(), "text/html");
});

app.Run();

public class StudentRow
{
    public int StudentId { get; set; }
    public string FirstName { get; set; } = "";
    public string LastName { get; set; } = "";
    public string Email { get; set; } = "";
    public string CourseName { get; set; } = "";
    public int YearLevel { get; set; }
    public decimal? GPA { get; set; }
    public bool IsActive { get; set; }
}
```

Appendix B — Connection String Template

Copy this template and fill in your own values. The settings shown match Azure SQL best practices (encrypted channel, 30s timeout, validate certificate).

```
Server=tcp:<YOUR-SQL-SERVER>.database.windows.net,1433;Initial Catalog=dbwebapp;Persist Security Info=False;User ID=sqldba;Password=<YOUR-PASSWORD>;MultipleActiveResultSets=False;Encrypt=True;TrustServerCertificate=False;Connection Timeout=30;
```

Parameter explanation

Parameter	Meaning
Server	Fully qualified name of the SQL server, with port 1433
Initial Catalog	The specific database to connect to (dbwebapp)
User ID / Password	SQL authentication credentials
Encrypt=True	Encrypts all traffic between client and server (required by Azure SQL)
TrustServerCertificate=False	Validates the server certificate against a trusted CA (recommended)
MultipleActiveResultSets=False	Disables MARS — only needed for specific ORM scenarios
Connection Timeout=30	Seconds to wait before failing a connection attempt
Persist Security Info=False	Don't return the password in the connection object after opening

> ⚠ **WARNING**
> Never store the full connection string with a real password in source control. Commit only a template (with `<YOUR-PASSWORD>` placeholder) and inject the real value via Key Vault or App Settings at deploy time.

Appendix C — Alternative Deployment Methods

For completeness, here are other common ways to deploy .NET apps to App Service. You don't need any of these for the lab — they are for future exploration.

C.1 Azure CLI one-liner

After running dotnet publish and zipping per Section 11:

```
az login
az webapp deploy \
  --resource-group rg-web-apps-001 \
  --name my-app-001 \
  --src-path ./StudentDemoApp.zip \
  --type zip
```

C.2 Visual Studio Right-click Publish (Windows)

- Right-click the project in Solution Explorer → **Publish...**.
- Choose **Azure** → **Azure App Service (Linux)**.
- Select your existing `my-app-001` instance.
- Click **Finish**, then **Publish**.

C.3 GitHub Actions

Sample workflow (save as .github/workflows/deploy.yml in your repo):

```
name: Deploy to Azure Web App

on:
  push:
    branches: [ main ]

jobs:
  build-and-deploy:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Setup .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '10.0.x'

      - name: Build and publish
        run: dotnet publish -c Release -o ./publish

      - name: Deploy to Azure Web App
        uses: azure/webapps-deploy@v3
        with:
          app-name: my-app-001
          publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
          package: ./publish
```

You get the `AZURE_WEBAPP_PUBLISH_PROFILE` value from the portal: `my-app-001` → **Download publish profile** → paste the XML content as a GitHub secret.

C.4 Bicep + GitHub Actions (IaC + CI/CD)

The most production-ready pattern is to have:

- Infrastructure described in Bicep (what you built in Lab 1)
- Application code in the same repo
- One workflow that deploys Bicep (infra) and then the code, using OIDC federated credentials (no stored secrets)

> 💡 **TIP**
> This is the logical follow-up to this lab. Once students understand the portal-based flow, rebuilding the same stack as Bicep + GitHub Actions solidifies every concept and is a realistic simulation of how Azure work actually happens in enterprise teams.

Congratulations 🎉

You have now taken an Azure Cloud Fundamentals lab from an empty resource group all the way to a live, data-driven web application. You've seen the full stack:

- Infrastructure provisioning (Lab 1)
- Application scaffolding with the .NET CLI
- Working with NuGet packages
- Minimal API hosting in ASP.NET Core
- Connecting code to managed PaaS services (SQL + Blob Storage)
- The publish / package / deploy lifecycle
- App Service configuration — env vars, startup commands, restarts
- Structured debugging of common deployment failures

The journey from here is to replace each hand-crafted step with an automated, repeatable, secure equivalent: Bicep/Terraform for the infra, GitHub Actions for CI/CD, Key Vault + Managed Identity for secrets, and Application Insights for observability. That is what Lab 3 and beyond should look like.