GAPVelocity.ai

TECHNICAL GUIDE

From VB6 to Docker

The ultimate journey from legacy desktop to cloud-native container

1. Extinction is normal

Picture two dinosaurs watching a massive fireball rush toward Earth. One says, “Wonder what that is?” The other replies, “Ah, probably nothing.”

Consider Kodak — the company that invented the consumer camera industry. When the first digital cameras appeared, engineers recognized a “hair on fire” moment. Management ignored the warning signs, insisting digital was expensive, inferior and would never replace film.

Extinction-level change used to take a while: the first steam-powered automobile was built in 1769, yet cars didn’t eclipse horse-drawn buggies in the US until 1920. Today change is breakneck — the first smartphone appeared in 1994, the iPhone in 2007, and now most of the planet runs on mobile computing.

If you’re reading this, you likely have legacy code — a desktop application built with VB6, PowerBuilder or legacy WinForms. It’s old, massive and complex, with decades of business logic. It works, people use it, but it is a massive liability.

You’ve built a wonderful factory powered by steam while your competitors run on electricity. The engineers who know how to run your steam factory are retiring, and there’s a daily risk the boiler explodes. Replacing everything manually would bleed your budget dry.

You are faced with an existential threat. Here is how to survive.

2. The destination: Docker and the cloud

This guide takes a legacy VB6 application and converts it into a modern web application with RESTful endpoints, an Angular/Blazor client UI and an ASP.NET Core backend.

- **The GAPVelocity AI advantage:** we use **VELO**, our agentic AI modernization platform, to automatically transform the legacy VB6 code into a modern web architecture — avoiding the risks of a manual rewrite.
- **The cloud-native leap:** we place the modernized web app into a container using Docker for Windows, run it locally, then push the container to Microsoft Azure to run from the cloud.

Docker is the icing on the modernization cake. Continuous integration and continuous deployment (CI/CD) are the extinction-level changes sweeping software engineering — and Docker makes CI/CD easier, safer and more reliable.

3. What is Docker?

Docker is a suite of tools used to create, manage and run *containers*.

Containers resemble virtual machines (VMs): they are autonomous units with their own application infrastructure, can access network ports and run securely on any host — a Linux container on Windows, or vice versa.

But containers differ fundamentally: **they load only the bare minimum required to do the job.** A VM loads an entire guest operating system; a container gets only the OS services in that specific kernel instance. That makes containers lightweight, fast to boot, highly scalable and highly resistant to attack — and a container that runs locally is guaranteed to run identically in AWS, Azure or a private datacenter.

4. The Docker workflow: from desktop to Azure

For this walkthrough, we assume the VB6 application has already been modernized into an ASP.NET Core web application using **VELO**. The steps to containerize it:

1. Install Docker for Windows
2. Enable Docker support in Visual Studio
3. Create a Dockerfile
4. Build the container image
5. Run the container locally
6. Push the container to Azure

Step 1 — Install Docker for Windows

Download Docker Community Edition (CE) from the Docker Store. It requires Windows 10 64-bit Pro, Enterprise or Education.

- ***Technical note:*** Docker uses Microsoft's Hyper-V hypervisor. If you use VMware Workstation, enable Hyper-V in Windows settings (this temporarily disables VMware VMs).
- After installing, right-click the Docker tray icon and ensure **Windows containers** is selected — the default is Linux.

Step 2 — Enable Visual Studio support

Add Docker support to your modernized ASP.NET Core project: right-click the project in Solution Explorer, choose **Add**, then **Docker Support**. When prompted, select **Windows** as the target OS. This generates two files — a `.dockerignore` and a `Dockerfile`.

Step 3 — Configuring the Dockerfile

The generated Dockerfile is a template. Because our modernized app needs specific Windows Server Core and ASP.NET capabilities, we specify the correct *base image* from Docker Hub. A robust configuration:

```dockerfile
# Escape character definition for Windows paths
# escape=`

# Specify the base image
FROM microsoft/aspnet:windowsservercore-10.0.14393.693

# Set up PowerShell as the shell
SHELL ["powershell", "-Command", "$ErrorActionPreference = 'Stop';"]

# Clean up default IIS site and create our app directory
RUN Remove-Website -Name 'Default Web Site'; `
    New-Item -Path 'C:\web-app' -Type Directory; `
    New-Website -Name 'web-app' -PhysicalPath 'C:\web-app' -Port 80 -Force

# Expose port 80 for HTTP traffic
EXPOSE 80

# Copy the published application into the container
COPY /bin/Release/PublishOutput/ /web-app
```

In this workflow we build and publish the app via Visual Studio to a local `/PublishOutput/` directory; the Dockerfile simply copies those binaries into the container's `/web-app` directory.

Step 4 — Building the container image

With the Dockerfile ready, open PowerShell, navigate to your project directory and run:

```powershell
docker build -t web-app:latest .
```

- `-t web-app:latest` tags the image with name `web-app` and version `latest`.
- `.` tells Docker to use the current directory as the build context.

Docker pulls the base image from the cloud (a few minutes the first time) and executes each step in your Dockerfile.

Step 5 — Running the image locally

Verify the image with `docker images`, then spin up the container:

```
POWERSHELL

docker run -d web-app
```

The `-d` switch runs the container in the background and returns a unique container ID. Confirm it's running with `docker ps`, which lists active containers, their randomly generated names (e.g. `sleepy_bartik`) and exposed ports (e.g. `80/tcp`). To view the app, retrieve the container IP:

```
POWERSHELL

docker inspect -f '{{range .NetworkSettings.Networks}}{{.IPAddress}}{{end}}' sleepy_bartik
```

Navigate to that IP (e.g. `172.19.209.167`) in your browser and your modernized application loads seamlessly.

5. Pushing the container to Azure

Running locally is great for testing, but production belongs in the cloud. Azure Container Registry provides a private repository for your Docker images.

1. Log into the Azure Portal.
2. Search for **Container Registry** and click Create.
3. Provide a registry name (e.g. `gapvelocitydemo`), which generates a login server (`gapvelocitydemo.azurecr.io`).

Back in PowerShell, tag your local image with the Azure repository name:

```
POWERSHELL

docker tag web-app gapvelocitydemo.azurecr.io/repos/sks:latest
```

Then push the image to the cloud:

```
POWERSHELL

docker push gapvelocitydemo.azurecr.io/repos/sks:latest
```

Once uploaded, open your Azure Container Registry, select **Repositories**, find your image, click the ellipsis next to the `latest` tag and choose **Run Instance**. Azure provisions a public IP for your container.

Navigate to that public IP in your browser. Your application now runs securely and scalably in the cloud, completely disconnected from its legacy desktop origins — and because the container is identical to the one you ran on your laptop, you have a 100% guarantee it executes perfectly.

6. Conclusion

We began with something horrifying yet common: a fragile, unsupportable VB6 desktop application.

Using **VELO**, we transformed it into a modern web application (Angular / C#) with all original business logic intact. We then encapsulated the entire application into a Docker container, ran it locally and pushed it to Azure.

The reasons for moving off legacy desktop frameworks grow stronger every day. CI/CD pipelines, automated testing and scalable cloud hosting are the future of enterprise software, and containers make the transition secure, portable and reliable — but you must modernize the legacy codebase first.

Stop managing legacy risk. Let GAPVelocity AI guide your application out of the steam age and into the cloud-native future.

gapvelocity.ai