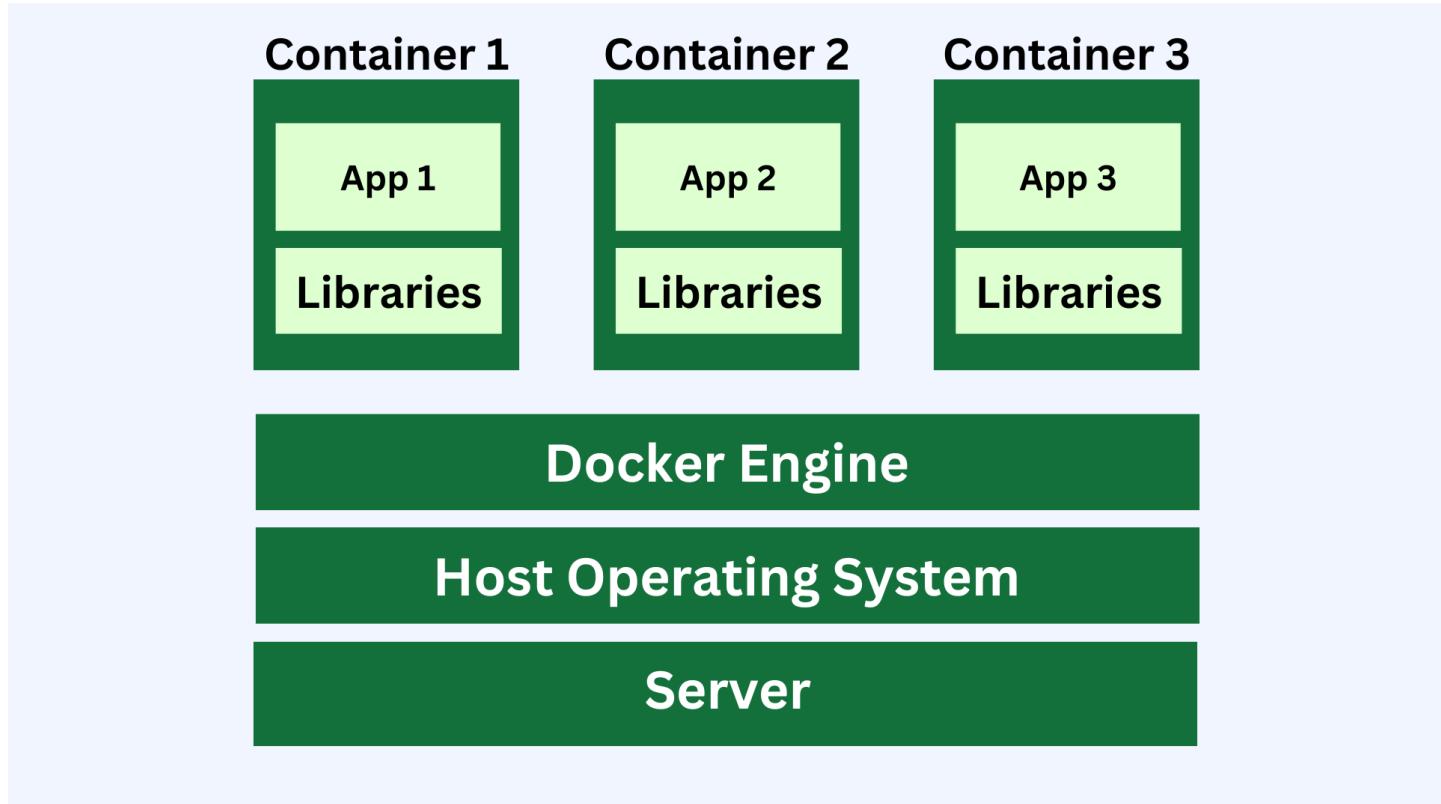


1. Giới thiệu



Hình 1. Cấu trúc tổng quát containerization

Trong bài lab này, mục tiêu là đóng gói (containerize) hai ứng dụng của Contoso gồm:

- Backend: ứng dụng Java (micro social media API)
- Frontend: ứng dụng .NET (UI)

Sau khi đóng gói thành các container image, hai ứng dụng sẽ được điều phối chạy cùng nhau thông qua Docker Compose để có thể triển khai nhất quán trên nhiều nền tảng.

Tài liệu tham khảo chính: docs/05-containerization.vi.md (dịch từ docs/05-containerization.md).

2. Bối cảnh và mục tiêu

2.1. Bối cảnh

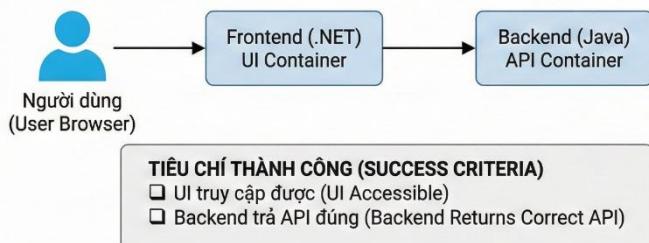
Contoso kinh doanh sản phẩm cho các hoạt động ngoài trời. Bộ phận marketing muốn triển khai một website mạng xã hội nhỏ để quảng bá sản phẩm. Hệ thống hiện có 2 phần tách rời:

- Backend Java: cung cấp API
- Frontend .NET: giao diện người dùng

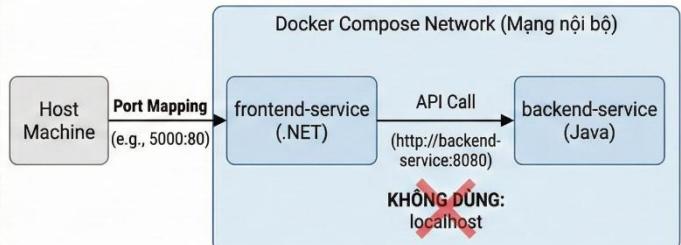
2.2. Mục tiêu

Trong bài lab này, cách tiếp cận là đi từ “bức tranh tổng thể” đến “cách các thành phần kết nối” và cuối cùng là “cách triển khai chạy thật” để có thể kiểm chứng được kết quả. Trước hết, xác định rõ hai thành phần chính của hệ thống: backend Java cung cấp API và frontend .NET cung cấp giao diện cho người dùng. Tiếp theo, làm rõ điểm người dùng truy cập (trình duyệt vào UI) và tiêu chí thành công (UI truy cập được, backend trả API đúng). Sau khi từng container chạy ổn khi chạy độc lập, hai dịch vụ được điều phối bằng Docker Compose để đảm bảo frontend gọi đúng backend thông qua mạng nội bộ của Docker (theo tên service), thay vì phụ thuộc vào localhost theo ngữ cảnh của từng container.

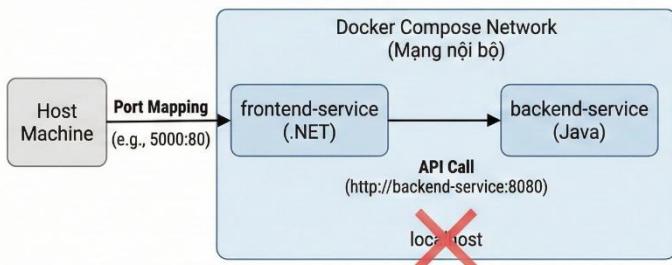
STAGE 1: BỨC TRanh TỔNG THỂ (BIG PICTURE)



STAGE 2: CÁCH CÁC THÀNH PHẦN KẾT NỐI (CONNECTION MECHANISM – DOCKER COMPOSE)



STAGE 2: CÁCH CÁC THÀNH PHẦN KẾT NỐI (CONNECTION MECHANISM - DOCKER COMPOSE)



STAGE 3: CÁCH TRIỂN KHAI CHẠY THẬT & KIỂM CHỨNG (ACTUAL DEPLOYMENT & VERIFICATION)



Các mục tiêu rõ ràng và cụ thể :

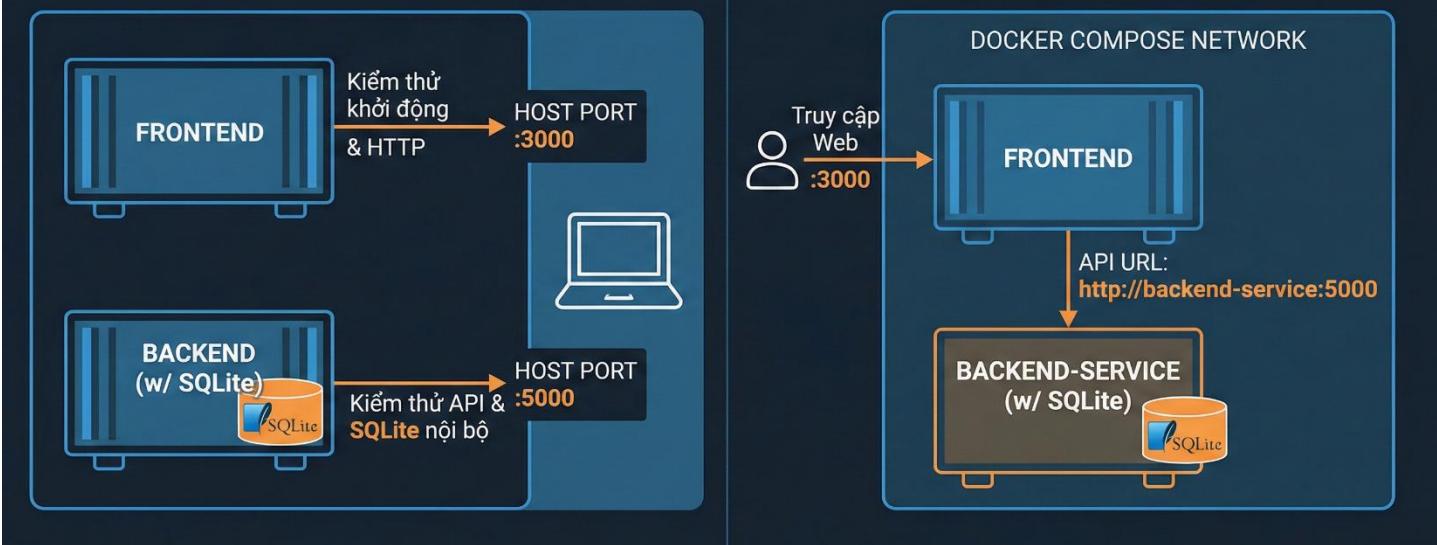
- Tạo Dockerfile cho backend Java và frontend .NET
- Build container image cho từng ứng dụng
- Chạy thử từng container để kiểm tra hoạt động
- Tạo file compose.yaml để điều phối chạy đồng thời 2 container
- Xác minh hệ thống chạy được bằng cách truy cập web (http://localhost:3030)

3. Tổng quan kiến trúc và luồng chạy

Về mặt vận hành, người dùng truy cập vào giao diện web (frontend), và frontend sẽ gọi các endpoint API của backend để lấy/ghi dữ liệu. Backend sử dụng SQLite làm nơi lưu trữ và file cơ sở dữ liệu được tạo ngay bên trong môi trường container để tránh phụ thuộc vào máy host. Khi chạy độc lập, việc kiểm thử tập trung vào khả năng khởi động dịch vụ và phản hồi HTTP qua các cổng được publish ra host. Khi chạy bằng Docker Compose, trọng tâm chuyển sang việc đảm bảo hai container “nhìn thấy nhau” và frontend trả đúng URL API của backend thông qua tên service trong network, giúp cấu hình chạy nhất quán và tái lập được.

CHẠY ĐỘC LẬP (STANDALONE TESTING)

DOCKER COMPOSE (INTEGRATION TESTING)



3.1. Thành phần

- Backend service (Java)
 - Lắng nghe cổng 8080 trong container
 - Khi chạy độc lập: ánh xạ host port 8080 → container port 8080
- Frontend service (.NET)
 - Lắng nghe cổng 8080 trong container
 - Khi chạy độc lập: ánh xạ host port 3030 → container port 8080

3.2. Biến môi trường quan trọng

- Backend Java
 - CODESPACE_NAME
 - GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN
 - Tạo file SQLite sns_api.db ngay trong image (không copy từ host)
- Frontend .NET
 - ApiSettingsBaseUrl: URL trả đến API backend (ví dụ: http://localhost:8080/api khi chạy độc lập)

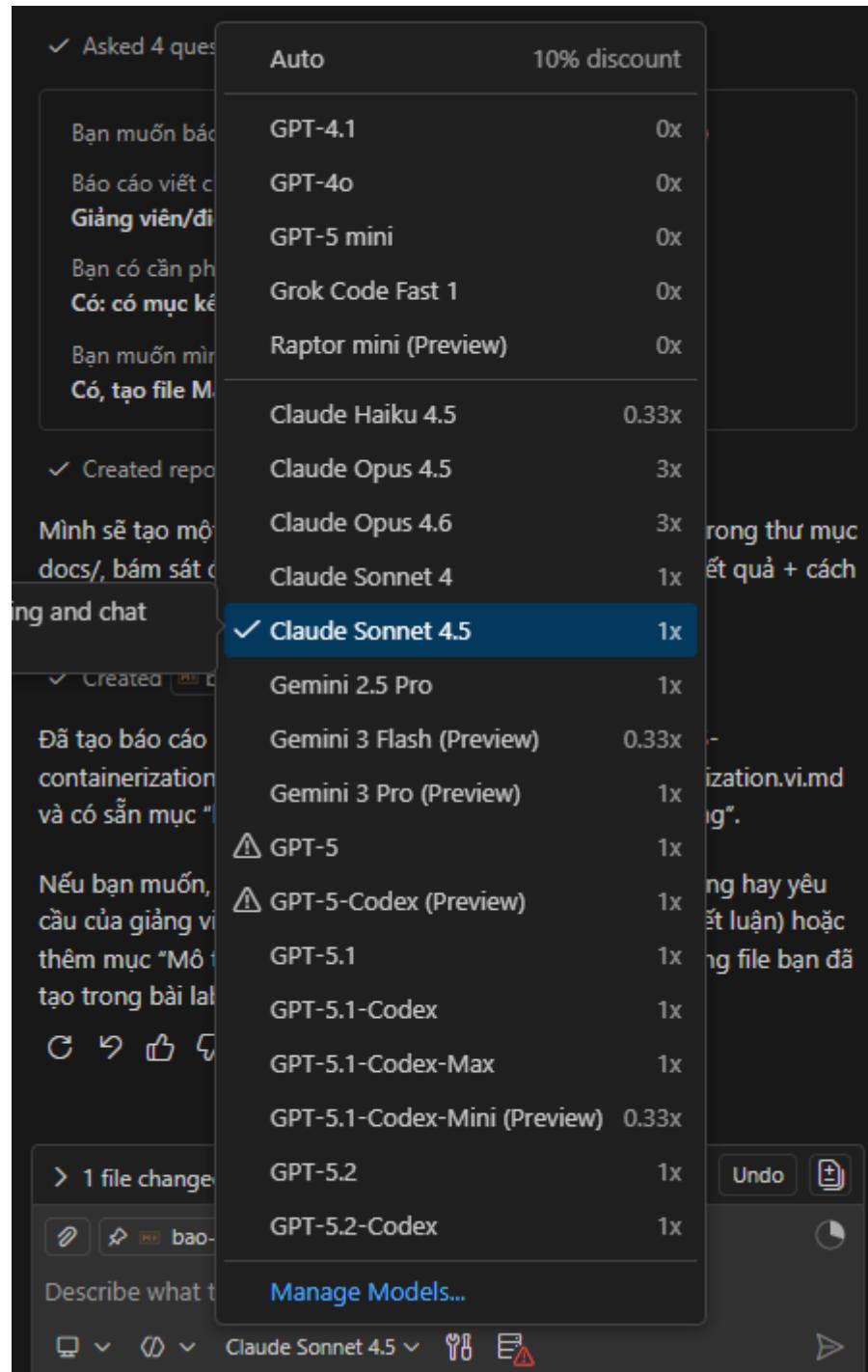
4. Chuẩn bị môi trường

4.1. Điều kiện trước

Thực hiện các bước chuẩn bị theo README ở thư mục gốc (đường dẫn tương đối từ docs/ là README.md).

4.2. Kiểm tra GitHub Copilot Agent Mode

- Mở GitHub Copilot trong VS Code / Codespaces
- Đảm bảo đang bật Agent Mode
- Chọn model: GPT-4.1 hoặc Claude Sonnet 4
- Đảm bảo đã cấu hình MCP Servers theo tài liệu setup: docs/00-setup.md



Hình 2. Model đã chọn

4.3. Chuẩn bị custom instructions

Thiết lập biến môi trường REPOSITORY_ROOT và copy custom

instructions cho bài containerization. Bash/Zsh:

```
REPOSITORY_ROOT=$(git rev-parse --show-toplevel)
cp -r $REPOSITORY_ROOT/docs/custom-instructions/containerization/. \
$REPOSITORY_ROOT/.github/
```

PowerShell:

```
$REPOSITORY_ROOT = git rev-parse --show-toplevel
Copy-Item -Path $REPOSITORY_ROOT/docs/custom-instructions/containerization/* ` 
-Destination $REPOSITORY_ROOT/.github/ -Recurse -Force
```

```
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-workshop'> $REPOSITORY_ROOT = git rev-parse --show-toplevel
>> Copy-Item -Path $REPOSITORY_ROOT/docs/custom-instructions/containerization/* ` 
>> -Destination $REPOSITORY_ROOT/.github/ -Recurse -Force
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-workshop'>'
```

Mục đích: giúp Copilot bám sát yêu cầu bài lab, tuân thủ tiêu chuẩn repo và giảm sai lệch khi sinh Dockerfile/compose.

5. Containerize ứng dụng Java (backend)

5.1. Phạm vi

- Source của ứng dụng Java nằm tại java/socialapp
- Tạo Dockerfile tên Dockerfile.java ở thư mục gốc repo

5.2. Yêu cầu kỹ thuật

- Dùng Microsoft OpenJDK 21
- Multi-stage build
- Trích xuất JRE từ JDK (giảm dung lượng image runtime)
- Expose/target port: 8080
- Pass biến môi trường CODESPACE_NAME và GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN từ host vào container
- Tạo file SQLite sns_api.db trong image (không copy từ máy host)

5.3. Prompt

```
- I'd like to build a container image of a Java app. Follow the instructions below.
-
-   - Identify all the steps first, which you're going to do.
-   - The Java app is located at `java/socialapp`.
-   - Your working directory is the repository root.
-   - Create a Dockerfile, `Dockerfile.java` .
-   - Use Microsoft OpenJDK 21.
-   - Use multi-stage build approach.
-   - Extract JRE from JDK.
-   - Use the target port number of `8080` for the container image.
-   - Add both environment variables, `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN` from the host to the container image.
-   - Create an SQLite database file, `sns_api.db` , in the container image. DO NOT Copy the file from the host.
```

CHAT

← BUILDING A DOCKER IMAGE FOR JAVA APPLICATION

✖ 05-containerization.md:71-82

- I'd like to build a container image of a Java app. Follow the instructions below.
-
- Identify all the steps first, which you're going to do.
- The Java app is located at `java/socialapp`.
- Your working directory is the repository root.
- Create a Dockerfile, `Dockerfile.java`.
- Use Microsoft OpenJDK 21.
- Use multi-stage build approach.
- Extract JRE from JDK.
- Use the target port number of `8080` for the container image.
- Add both environment variables, `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN` from the host to the container image.

Local Agent Claude Sonnet 4.5

✓ Analyzed Java app structure and planned Dockerfile creation

I'll help you build a container image for the Java app. Let me first identify all the steps and examine the Java application structure.

Steps I'll perform:

1. Examine the Java app structure at `java/socialapp` to understand the build configuration
2. Create a multi-stage `Dockerfile.java` that:
 - o Stage 1: Build the Java app using Microsoft OpenJDK 21 with Gradle
 - o Stage 2: Extract JRE from JDK using jlink
 - o Stage 3: Create runtime image with extracted JRE
 - o Create SQLite database file `sns_api.db`
 - o Configure environment variables `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN`
 - o Expose port 8080

Let me start by examining the Java app structure:

✓ Reviewed socialapp file

Let me check the complete directory to find the Java app:

✓ Reviewed 4 files

> 1 file changed +69 -1

Keep

Undo



✖ 05-containerization.md:71-82

Describe what to build next

Local Agent Claude Sonnet 4.5

Hình 3. Prompt cho agent để đóng gói backend java

CHAT

← BUILDING A DOCKER IMAGE FOR JAVA APPLICATION

✓ Analyzed Java app structure and planned Dockerfile creation

I'll help you build a container image for the Java app. Let me first identify all the steps and examine the Java application structure.

Steps I'll perform:

1. Examine the Java app structure at `java/socialapp` to understand the build configuration
2. Create a multi-stage `Dockerfile.java` that:
 - Stage 1: Build the Java app using Microsoft OpenJDK 21 with Gradle
 - Stage 2: Extract JRE from JDK using `jlink`
 - Stage 3: Create runtime image with extracted JRE
 - Create SQLite database file `sns_api.db`
 - Configure environment variables `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN`
 - Expose port 8080

Let me start by examining the Java app structure:

✓ Reviewed socialapp file

Let me check the complete directory to find the Java app:

✓ Reviewed 4 files

Now I'll create the multi-stage `Dockerfile.java` at the repository root based on the Java app structure:

✓ Created Dockerfile.java

Perfect! I've created the `Dockerfile.java` at the repository root with all the required specifications:

Summary of what was created:

Multi-stage Dockerfile with 3 stages:

1. **Stage 1 (builder):** Uses Microsoft OpenJDK 21 to build the Java Spring Boot application with Gradle
 - Copies Gradle wrapper and build files
 - Downloads dependencies
 - Builds the application JAR
2. **Stage 2 (jre-builder):** Extracts a custom JRE from JDK using `jlink`
 - Creates a minimal JRE with only necessary modules
 - Reduces final image size by excluding unnecessary JDK components
3. **Stage 3 (runtime):** Creates the final runtime image

Hình 4. Phản hồi từ agent sau khi prompt backend java

5.4. Build image và kiểm tra

Prompt build image:

Use `Dockerfile.java` and build a container image.

- Use `contoso-backend` as the container image name.
- Use `latest` as the container image tag.
- Verify if the container image is built properly.
- If the build fails, analyze the issues and fix them.

Hình 5. Prompt dùng `dockerfile.java` để build image

- Use `Dockerfile.java` and build a container image.
- - Use `contoso-backend` as the container image name.
 - Use `latest` as the container image tag.
 - Verify if the container image is built properly.
 - If the build fails, analyze the issues and fix them.

Được rồi, để tôi build container image từ `Dockerfile.java` với tên `contoso-backend:latest`:

```
① docker build -f Dockerfile.java -t contoso-backend:latest .
18 |
19 |     # Build the application
-----
✓      ERROR: failed to build: failed to solve: failed to compute cache key: fail
calculate checksum of ref nnyfjdwgua77veb43ibnm74t::b5op3pdtkqtq6xqwzafcx
"/java/socialapp/src": not found

View build details: docker-desktop://dashboard/build/desktop-linux/desktop/85wwkv3u9vte11so920vetsi1
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-wor
```

✓ Updated Dockerfile.java to correct path reference

Có lỗi xảy ra! Dockerfile đang tìm source code tại `java/socialapp` nhưng app thực sự nằm ở `complete/java/socialapp`. Để tôi sửa lại Dockerfile.java:

✓ Updated Dockerfile.java

Đã sửa xong path. Bây giờ thử build lại:

```
✓ • docker build -f Dockerfile.java -t contoso-backend:latest .
```

Hình 6. Phản hồi từ agent sau khi nhập prompt build image từ `dockerfile.java`

Kết quả kỳ vọng:

- Image contoso-backend:latest build thành công
- Không còn lỗi Gradle/build
- Container có thể khởi động và lắng nghe cổng 8080

	Name	Tag	Image ID	Created	Size	Actions
<input type="checkbox"/>	contoso-backend	latest	4996b6198c24	16 hours ago	503.74 MB	
<input type="checkbox"/>	contoso-frontend	latest	534bf612a726	2 minutes ago	345.74 MB	

Hình 7. Các image đã được build thành công

5.5. Chạy container và xác minh

Prompt chạy container:

Restore Checkpoint

- Use the container image just built, run a container and verify if the app is running properly.
- - Use the host port of `8080`.
 - Both `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN` values

05-containerization.md:103-106

Hình 8. Prompt chạy container

Được rồi, để tôi chạy container backend Java với cổng 8080. Trước tiên để tôi kiểm tra xem có đang trong môi trường GitHub Codespaces không:

```
✓ • if ($env:CODESPACE_NAME) { Write-Host  
"CODESPACE_NAME=$env:CODESPACE_NAME" } else { Write-Host "Không phải  
GitHub Codespaces environment" }; if  
($env:GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN) { Write-Host  
"GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN=$env:GITHUB_CODESPACES_PORT_FO  
RWARDING_DOMAIN" } else { Write-Host "Không có  
GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN" }
```

✓ Ran container without passing variables

Môi trường hiện tại không phải GitHub Codespaces, vì vậy các biến môi trường này không tồn tại. Tôi sẽ chạy container backend với cổng 8080:

```
✓ • docker run -d --name contoso-backend-test -p 8080:8080 contoso-  
backend:latest
```

✓ Started container and checked its status

Tuyệt! Container đã khởi chạy. Để tôi kiểm tra trạng thái và logs:

```
✓ • Start-Sleep -Seconds 3; docker ps -a --filter name=contoso-backend-test  
--format "table {{.ID}}\t{{.Names}}\t{{.Status}}\t{{.Ports}}"
```

✓ Checked logs to ensure Spring Boot app started successfully

```
✓ • docker logs contoso-backend-test --tail 30
```

✓ Spring Boot application khởi động thành công và kiểm tra API

Xuất sắc! Ứng dụng đã khởi động thành công. Để tôi kiểm tra API endpoint:

```
✓ • Start-Sleep -Seconds 2; curl.exe -s  
http://localhost:8080/actuator/health | ConvertFrom-Json | ConvertTo-  
Object
```

Hình 9. Phản hồi từ agent

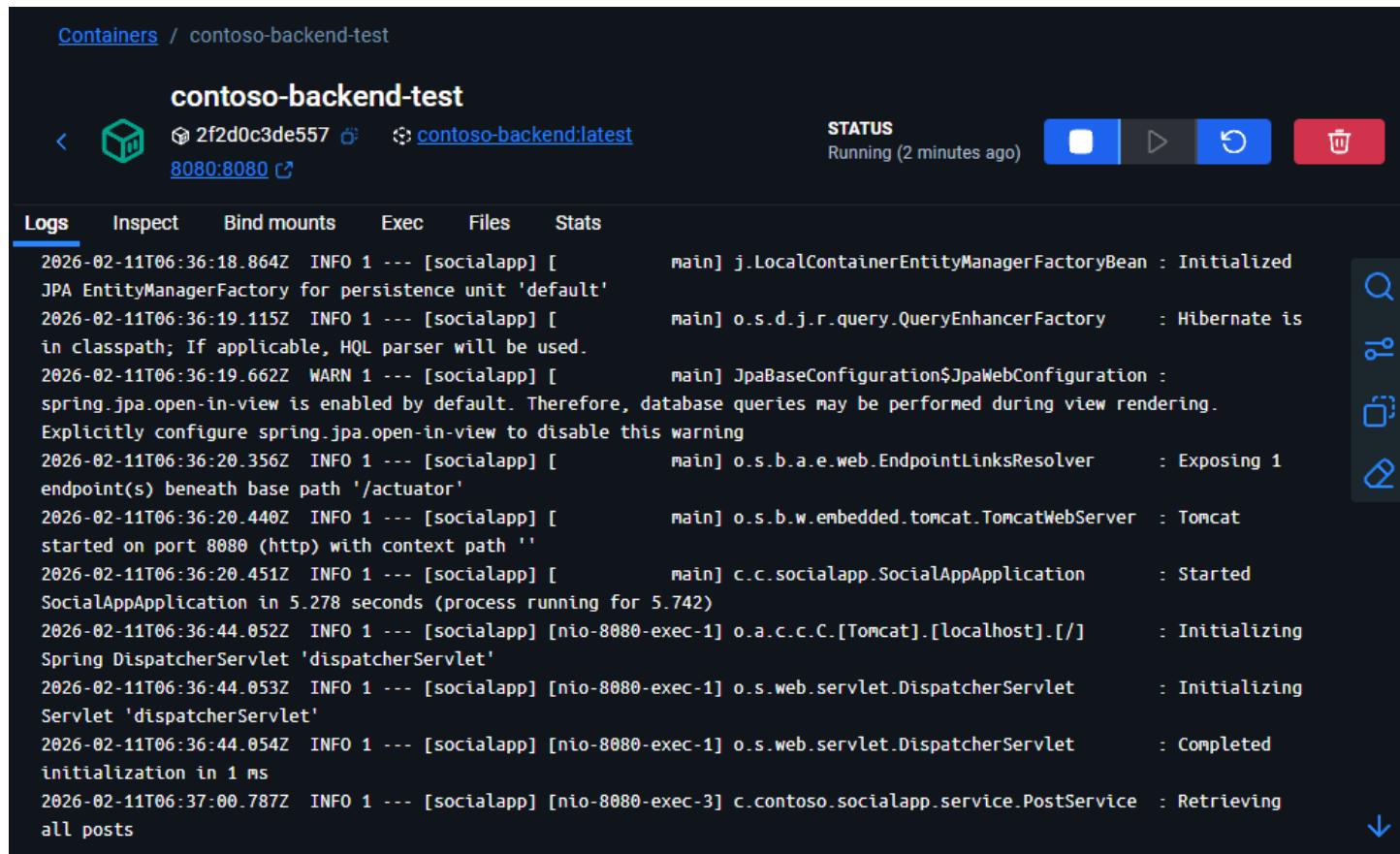
- Use the container image just built, run a container and verify if the app is running properly.
- Use the host port of `8080`.
- Both `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN` values should be the ones from GitHub Codespaces.

- Kiểm tra container đang chạy (docker ps)

```
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-workshop'>'docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS
TS
2f2d0c3de557 contoso-backend:latest "java -jar app.jar" About a minute ago Up About a minute 0.0
.0.0:8080->8080/tcp, [::]:8080->8080/tcp contoso-backend-test
a4d326e33abe contoso-frontend:latest "dotnet Contoso.Blaz..." 13 minutes ago Up 13 minutes 0.0
.0.0:3030->8080/tcp, [::]:3030->8080/tcp contoso-frontend-test
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-workshop'>'
```

Hình 10. Docker ps

- Truy cập thử endpoint (tùy API), hoặc kiểm tra log container để thấy ứng dụng start thành công



Hình 11. Log container backend

6. Containerize ứng dụng .NET (frontend)

6.1. Phạm vi

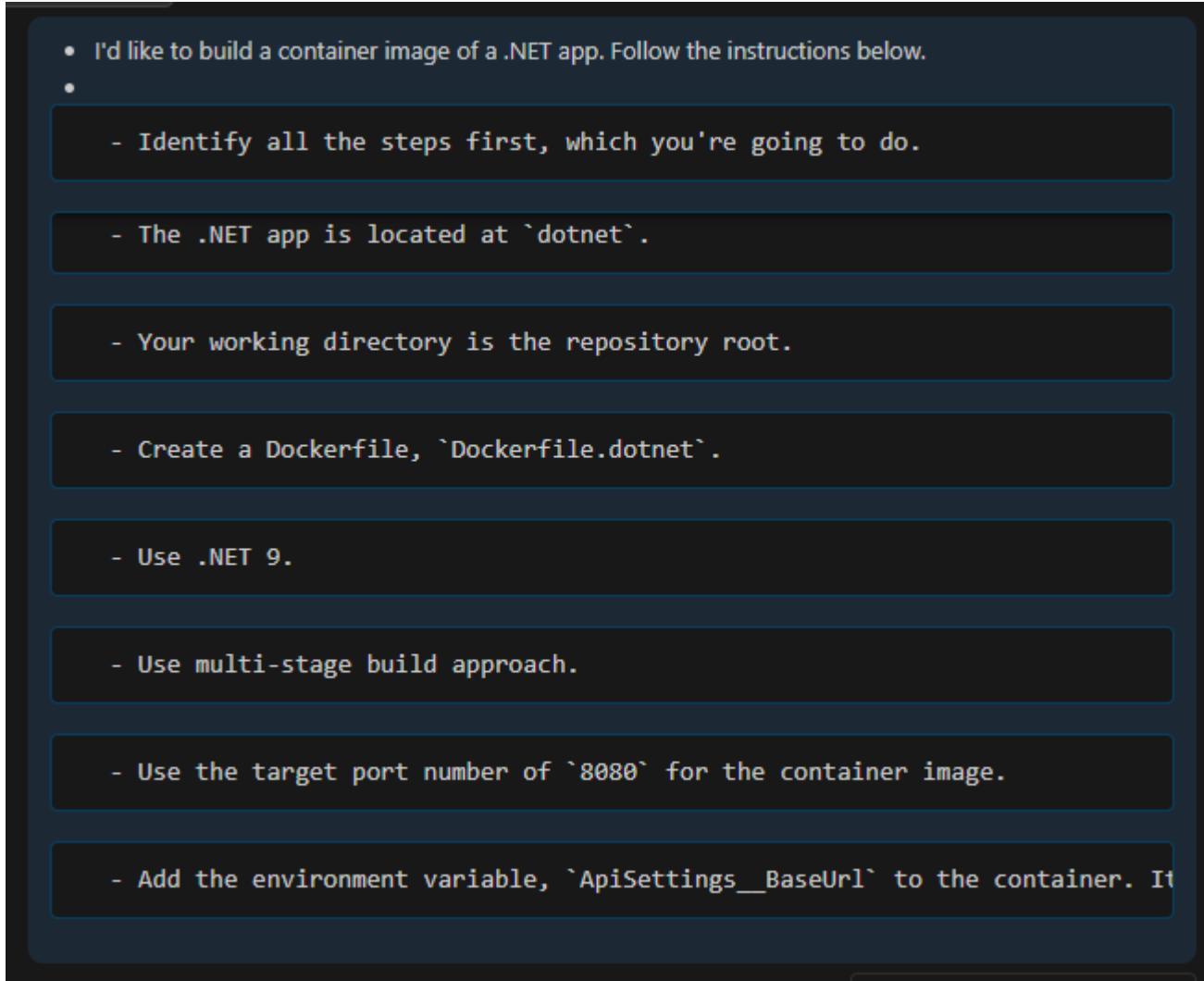
- Source của ứng dụng .NET nằm tại dotnet
- Tạo Dockerfile tên Dockerfile.dotnet ở thư mục gốc repo

6.2. Yêu cầu kỹ thuật

- Dùng .NET 9
- Multi-stage build
- Expose/target port: 8080
- Thiết lập biến môi trường ApiSettingsBaseUrl trả về backend Java, ví dụ: http://localhost:8080/api (khi chạy độc lập)

6.3.Prompt

- I'd like to build a container image of a .NET app. Follow the instructions below.
- - Identify all the steps first, which you're going to do.
 - The .NET app is located at `dotnet`.
 - Your working directory is the repository root.
 - Create a Dockerfile, `Dockerfile.dotnet`.
 - Use .NET 9.
 - Use multi-stage build approach.
 - Use the target port number of `8080` for the container image.
 - Add the environment variable, `ApiSettings__BaseUrl` to the container. It should point to the Java app, `http://localhost:8080/api`.



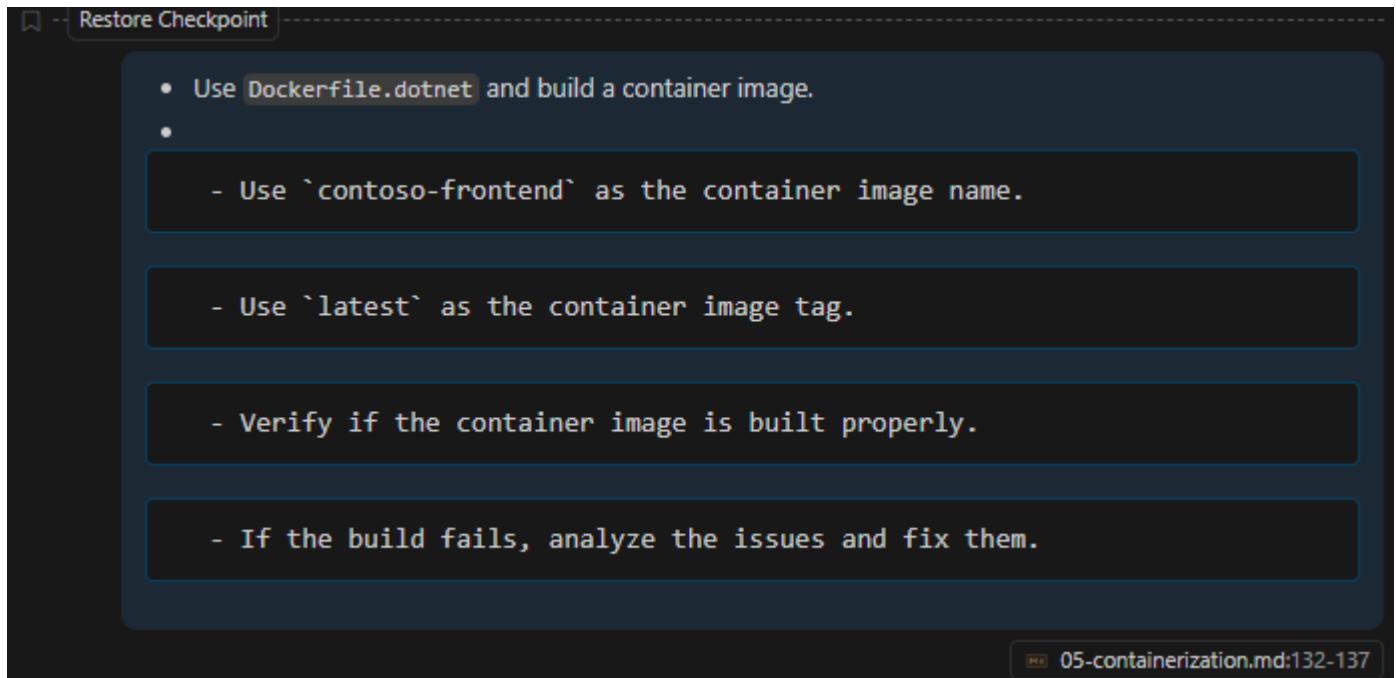
Hình 12. Prompt tạo dockerfile.dotnet

6.4.Build image và kiểm tra

Prompt build image:

Hãy dùng `Dockerfile.dotnet` để build một container image.

- Dùng `contoso-frontend` làm tên container image.
- Dùng `latest` làm tag của container image.
- Xác minh container image được build đúng.
- Nếu build thất bại, hãy phân tích nguyên nhân và sửa.



Hình 13. prompt build image dockerfile.dotnet

✓ Checked for Dockerfile.dotnet existence and planned creation

I'll build the container image using `Dockerfile.dotnet`. Let me first check if the Dockerfile exists, then build the image.

✓ Searched for files matching Dockerfile.dotnet and checked documentation

I see that `→ Dockerfile.dotnet` doesn't exist yet in the repository root. According to the workshop flow, I need to create it first. Let me examine the .NET app structure and create the Dockerfile, then build the image.

✓ Reviewed 4 files

I'll create the `→ Dockerfile.dotnet` first based on the requirements, then build the container image.

✓ Created Dockerfile.dotnet

Now let me build the container image. But first, I need to verify the actual location of the .NET app:

✓ Searched for files matching Contoso.BlazorApp.csproj

I need to update the Dockerfile to point to the correct location where the .NET app actually exists (complete/dotnet):

✓ Updated Dockerfile.dotnet

Now let me build the container image:

```
① docker build -f Dockerfile.dotnet -t contoso-frontend:latest .
[+] Building 0.1s (1/1) FINISHED
    => [internal] load build definition from Dockerfile.dotnet
    => => transferring dockerfile: 2B
ERROR: failed to build: failed to solve: failed to read dockerfile: open Dockerfile.dotnet: no such file or directory

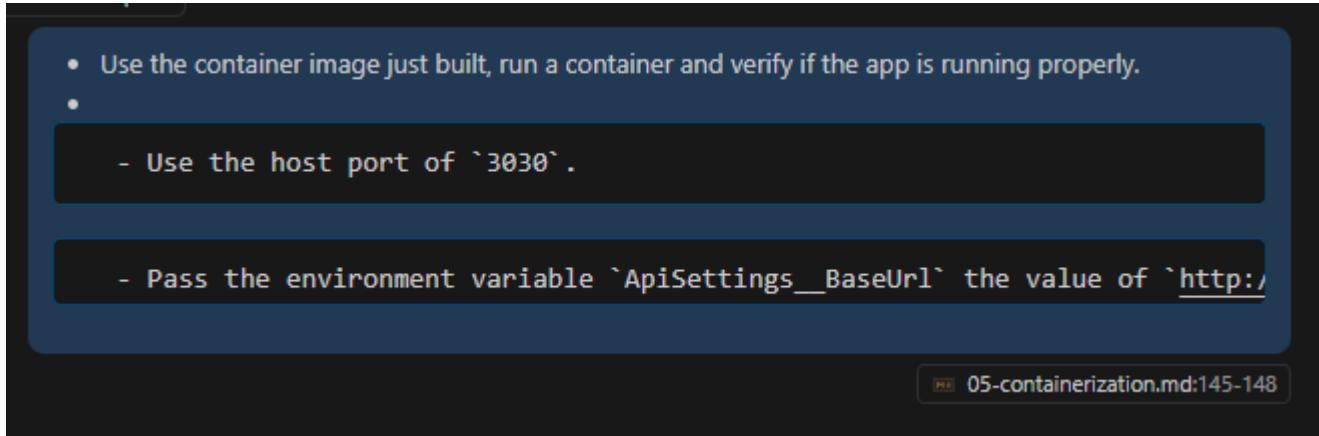
View build details: docker-desktop://dashboard/build/desktop-linux/desktop/k9bq8gsj32b9wgn cqgunljwju
```

Hình 14. Phản hồi từ agent sau khi prompt build image

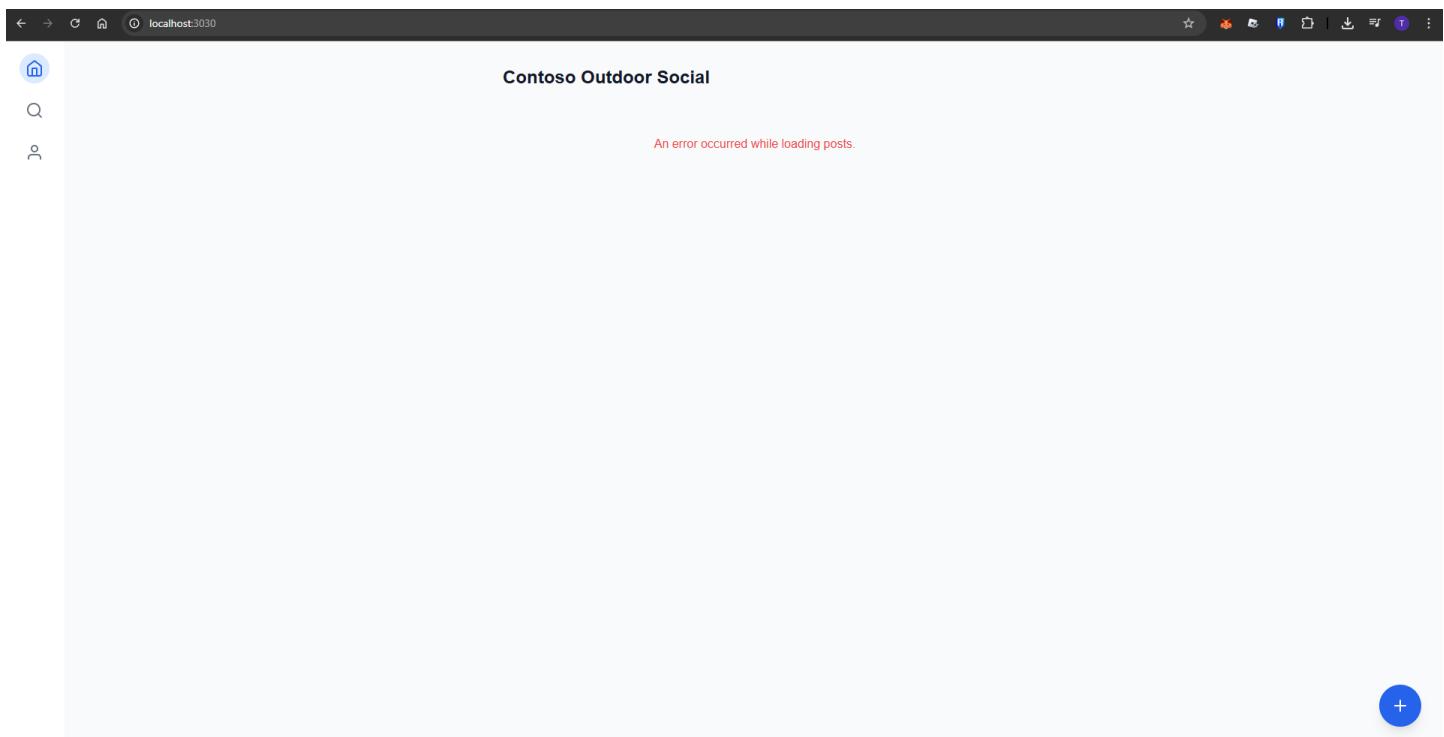
6.5. Chạy container và xác minh

Prompt chạy container:

- Use the container image just built, run a container and verify if the app is running properly.
- - Use the host port of `3030`.
 - Pass the environment variable ApiSettings__BaseUrl the value of `http://localhost:8080/api`.



Hình 15. Prompt chạy container frontend dotnet



Hình 16. UI frontend

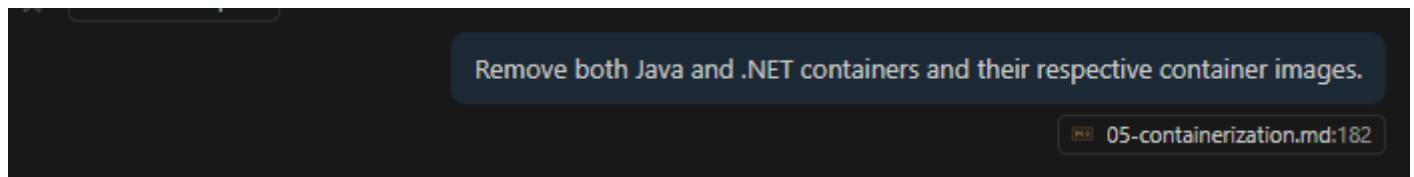
Lúc này vì còn chạy các container riêng biệt nên chưa kết nối được với backend dẫn tới chỉ có UI. Phần sau sẽ đề cập đến việc docker compose để chạy chung.

6.6. Dọn dẹp tài nguyên

Mục đích: đảm bảo trạng thái sạch trước khi sang

```
Remove both Java and .NET containers and their respective container images.
```

bước Docker Compose. Prompt tham khảo:



Hình 17. Prompt dọn dẹp tài nguyên

```
✓ Stopped and removed containers and images

Được rồi, để tôi xóa cả containers và images của Java và .NET:

✓ • docker stop contoso-backend-test contoso-frontend-app
✓ Removed stopped containers
✓ • docker rm contoso-backend-test contoso-frontend-app
✓ Deleted containers and planned to remove container images

• docker rmi contoso-backend:latest contoso-frontend:latest
Untagged: contoso-backend:latest
Deleted: sha256:4e48f13f6ce23a0bd13de545135430c29cb776ba779e94f57b5548ca9e
Error response from daemon: conflict: unable to delete contoso-frontend:latest (must be forced) - container a4d326e33abe is using its referenced image 534726
TRADUCTOAN@TRADUCTOAN C:\Seminar\Vibecoding\github-copilot-vibe-coding-workspace

✓ Deleted backend image, encountered frontend image error
✓ • docker rm -f a4d326e33abe 2>$null; docker rmi contoso-frontend:latest
✓ Verified all containers and images were deleted

• Write-Host "==== Containers còn lại ===" ; docker ps -a --filter "name=contoso" --format "table {{.Names}}\t{{.Image}}\t{{.Status}}"
Write-Host "`n==== Images còn lại ===" ; docker images --filter "reference=contoso-*" --format "table {{.Repository}}\t{{.Tag}}\t{{.Size}}"
✓ Removed all containers and images successfully
```

7. Điều phối các container bằng Docker Compose

7.1. Mục tiêu

- Tạo compose.yaml để chạy cả backend và frontend cùng một network
- Thiết lập biến môi trường và mapping cổng đúng yêu cầu

7.2. Yêu cầu

- Backend:
 - Dockerfile.java
 - container name: contoso-backend
 - ports: 8080:8080
 - network: contoso
 - pass CODESPACE_NAME, GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN
- Frontend:
 - Dockerfile.dotnet
 - container name: contoso-frontend
 - ports: 3030:8080
 - network: contoso
 - ApiSettingsBaseUrl trỏ đến /api của backend

7.3. Prompt để tạo compose.yaml

```
- I'd like to create a Docker Compose file. Follow the instructions below.  
-  
-     - Identify all the steps first, which you're going to do.  
-     - Your working directory is the repository root.  
-     - Use `Dockerfile.java` as a backend app.  
-     - Use `Dockerfile.dotnet` as a frontend app.  
-     - Create `compose.yaml` as the Docker Compose file.  
-     - Use `contoso` as the network name.  
-     - Use `contoso-backend` as the container name of the Java app. Its target port is 8080, and host port is 8080.  
-     - Use `contoso-frontend` as the container name of the .NET app. Its target port is 8080, and host port is 3030.  
-     - Add both environment variables, `CODESPACE_NAME` and `GITHUB_CODESPACES_PORT_FORWARDING_DOMAIN` from the host to the Java container.  
-         - Add the environment variable, `ApiSettingsBaseUrl` to the .NET container. It should point to the Java app's `/api`.
```

- I'd like to create a Docker Compose file. Follow the instructions below.
- - Identify all the steps first, which you're going to do.
- Your working directory is the repository root.
- Use `Dockerfile.java` as a backend app.
- Use `Dockerfile.dotnet` as a frontend app.
- Create `compose.yaml` as the Docker Compose file.
- Use `contoso` as the network name.
- Use `contoso-backend` as the container name of the Java app. Its target port is 8080.
- Use `contoso-frontend` as the container name of the .NET app. Its target port is 80.
- Add both environment variables, `CODESPACE_NAME` and `GITHUB_CODESPACES_F
- Add the environment variable, `ApiSettings__BaseUrl` to the .NET container.

[05-containerization.md:182]

Hình 18. Prompt tạo docker compose

✓ Planned Docker Compose file creation steps

Được rồi! Để tôi tạo Docker Compose file. Đây là các bước tôi sẽ thực hiện:

Các bước:

1. Tạo file `compose.yaml` tại repository root
2. Định nghĩa network "contoso" để các containers giao tiếp với nhau
3. Cấu hình backend service (Java):
 - o Build từ `Dockerfile.java`
 - o Container name: `contoso-backend`
 - o Port mapping: `8080:8080`
 - o Environment variables từ host
4. Cấu hình frontend service (.NET):
 - o Build từ `Dockerfile.dotnet`
 - o Container name: `contoso-frontend`
 - o Port mapping: `3030:8080`
 - o `ApiSettings__BaseUrl` trỏ đến backend service qua Docker network
5. Kết nối cả hai services vào network "contoso"

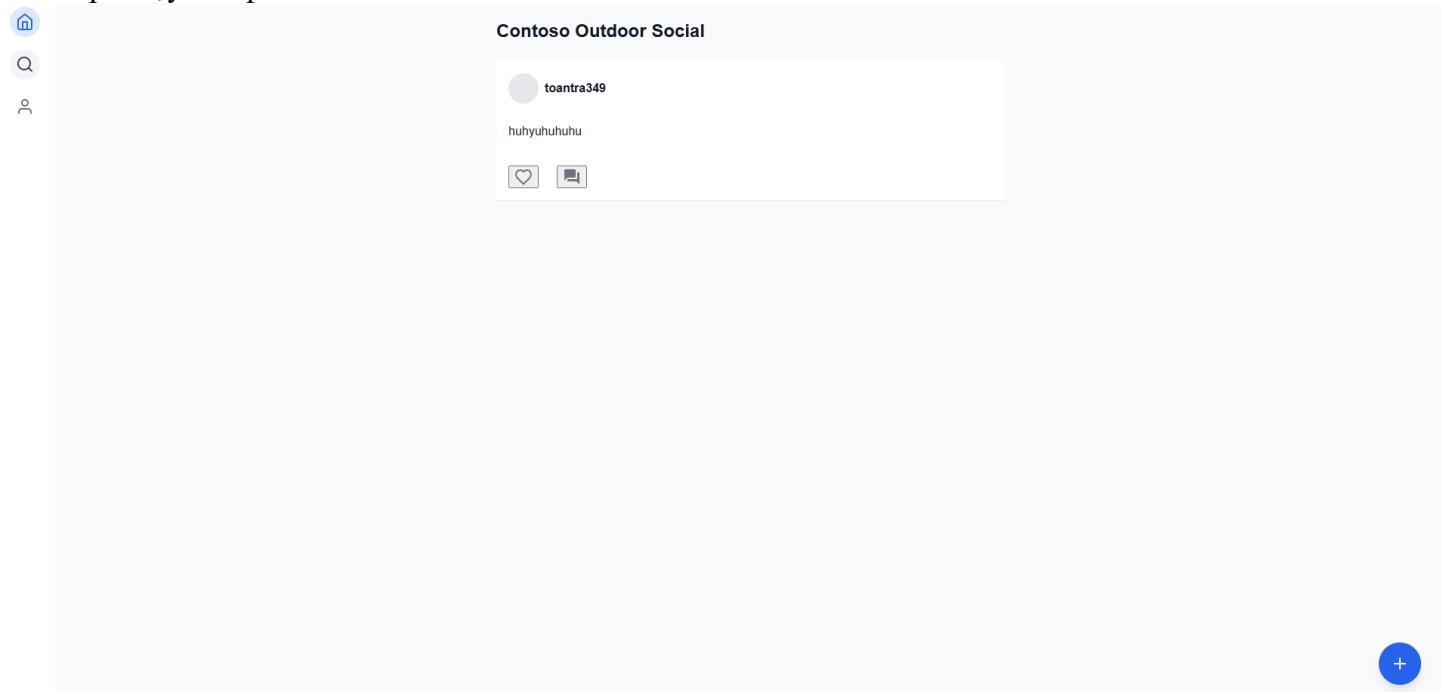
Bây giờ tôi tạo file:

✓ Created `compose.yaml`

Hoàn thành! Đã tạo file `compose.yaml` với cấu hình đầy đủ. Đây là những gì đã được thiết lập:

7.4. Chạy compose và xác minh

Prompt chạy compose:



Hình 19. Web đã chạy hoàn chỉnh backend và frontend

Run the Docker compose file and verify if all the apps are running properly.

The screenshot shows the Docker Compose interface with two containers listed:

- contoso-backend**:
IP: github-copilot-vibe-c...8080
Logs:

```
System.Net.Http.HttpClient.Default.ClientHandler[101]
Received HTTP response headers after 124.4162ms -
201
info:
System.Net.Http.HttpClient.Default.LogicalHandler[101]
End processing HTTP request after 124.5713ms - 201
info:
System.Net.Http.HttpClient.Default.LogicalHandler[100]
Start processing HTTP request GET http://contoso-backend:8080/api/posts
info:
System.Net.Http.HttpClient.Default.ClientHandler[100]
Sending HTTP request GET http://contoso-backend:8080/api/posts
```
- contoso-frontend**:
IP: github-copilot-vibe-c...3030
Logs:

```
2026-02-11T06:50:29.539Z INFO 1 --- [socialapp] [nio-8080-exec-6] c.contoso.socialapp.service.PostService : Retrieving all posts
info:
System.Net.Http.HttpClient.Default.ClientHandler[101]
Received HTTP response headers after 17.4587ms -
200
info:
System.Net.Http.HttpClient.Default.LogicalHandler[101]
End processing HTTP request after 17.6057ms - 200
```

Hình 20. Container sau khi docker compose

8. Kết quả đạt được

Sau khi hoàn thành, hệ thống đạt các kết quả kỳ vọng:

- Tạo được Dockerfile cho Java và .NET
- Build thành công 2 image: contoso-backend:latest và contoso-frontend:latest
- Chạy được 2 container độc lập để kiểm tra
- Tạo được compose.yaml để điều phối và chạy đồng thời
- Xác minh UI truy cập từ trình duyệt ở <http://localhost:3030>

9. Nhận xét và bài học rút ra

- Multi-stage build giúp giảm kích thước image runtime, tách bước build và chạy.
- Việc tách backend/frontend thành container độc lập giúp triển khai linh hoạt và nhất quán.
- Docker Compose giúp chuẩn hoá cấu hình chạy nhiều service: network, port mapping, env vars.
- Cần đặc biệt chú ý biến môi trường endpoint API và cấu hình networking khi chạy nhiều container.