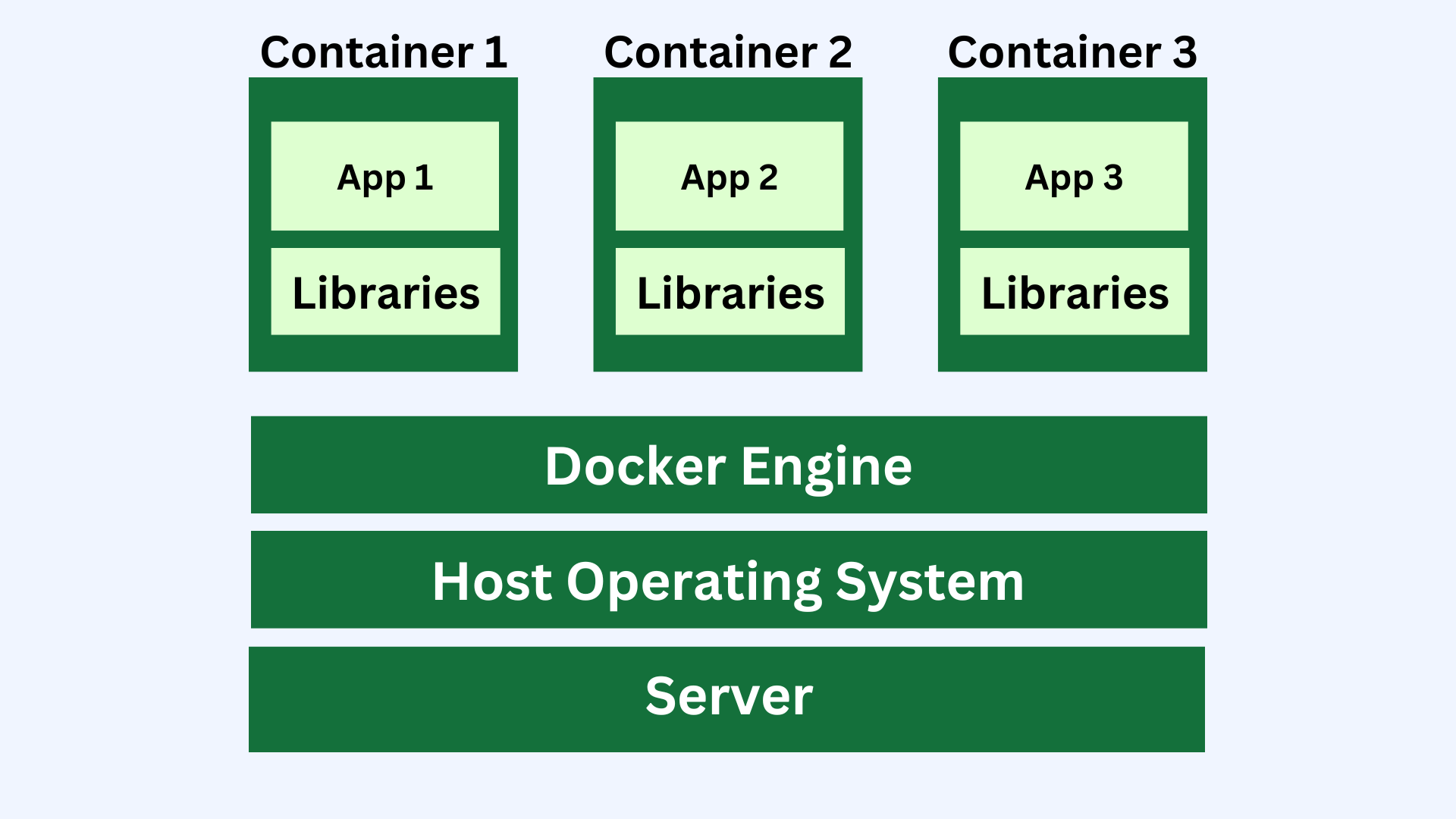
Báo cáo: Đóng gói ứng dụng bằng container (05 – Containerization )

# 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](file:///C:\tmp\docs\05-containerization.vi.md) (dịch từ [docs/05-containerization.md](file:///C:\tmp\docs\05-containerization.md)).

# Bối cảnh và mục tiêu

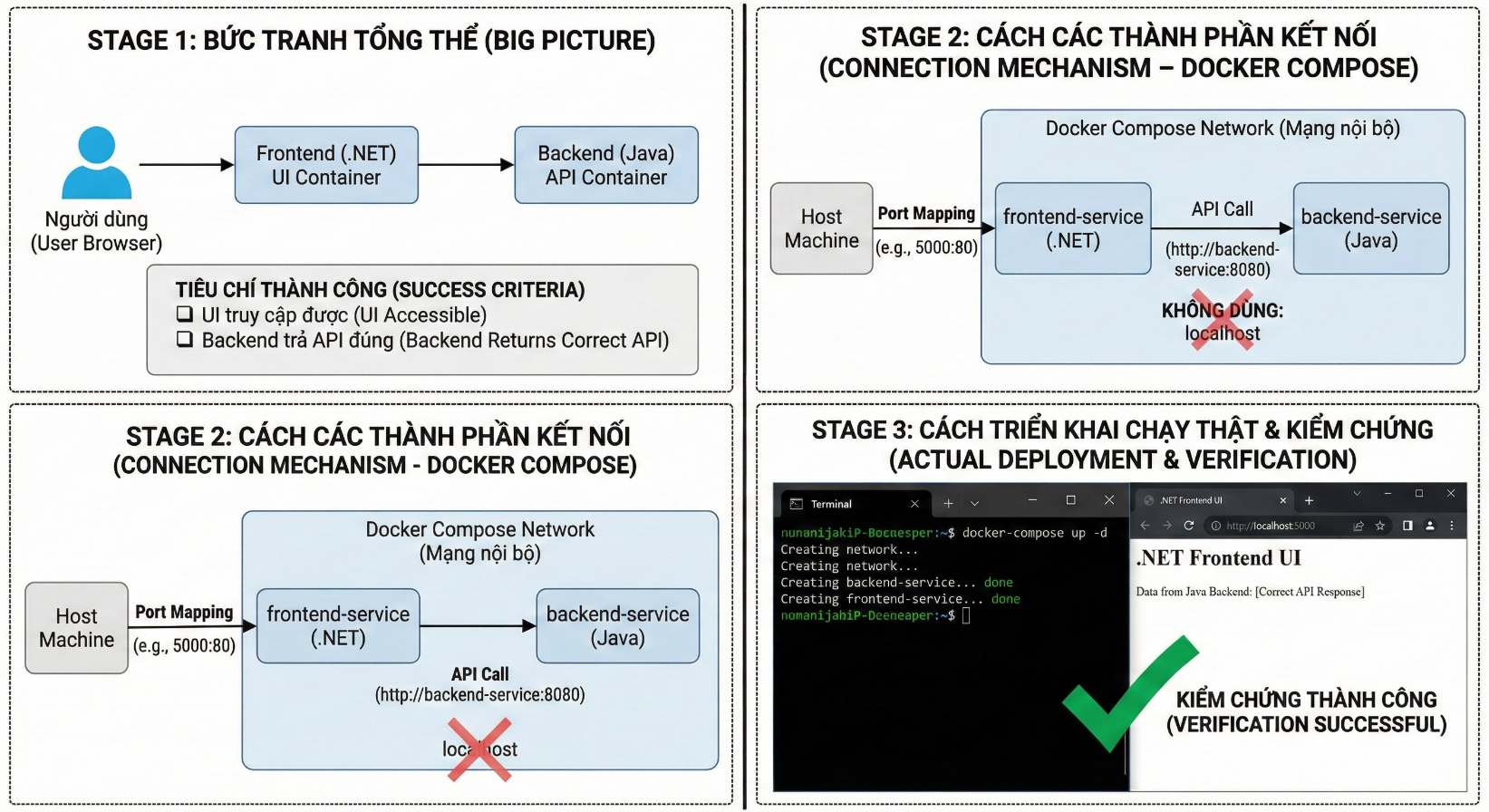
## Bối cảnh

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

## 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.

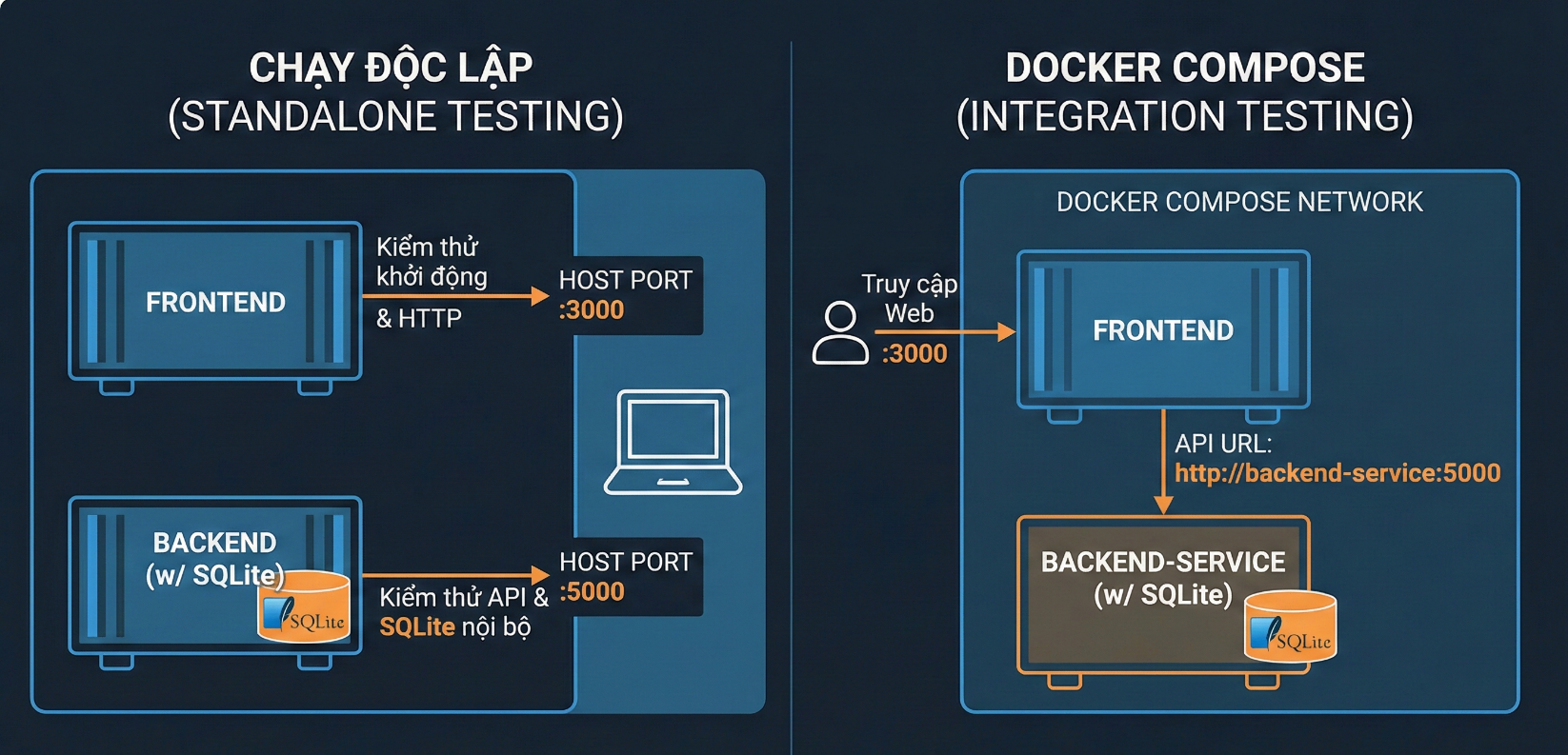


Các mục tiêu rõ ràng và cụ thể :

* + - 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](http://localhost:3030/))

# 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 trỏ đúng URL API của backend thông qua tên service trong network, giúp cấu hình chạy nhất quán và tái lập được.



## 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

## 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 trỏ đến API backend (ví dụ: <http://localhost:8080/api> khi chạy độc lập)

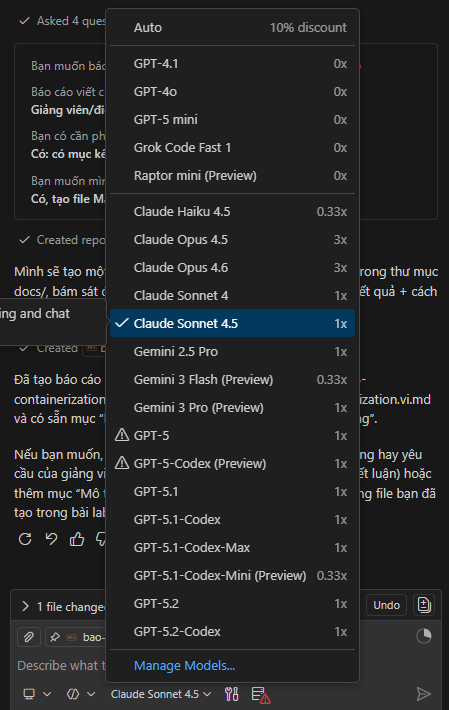
# Chuẩn bị môi trường

## Điều kiện trước

Thực hiện các bước chuẩn bị theo [README](file:///C:\tmp\README.md) ở thư mục gốc (đường dẫn tương đối từ docs/ là [README.md](file:///C:\tmp\README.md)).

## 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](file:///C:\tmp\docs\00-setup.md)



Hình 2. Model đã chọn

## 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/. \

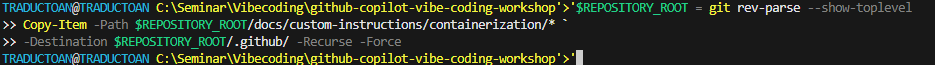
$REPOSITORY\_ROOT/.github/

PowerShell:

$REPOSITORY\_ROOT = git rev-parse --show-toplevel

Copy-Item -Path $REPOSITORY\_ROOT/docs/custom-instructions/containerization/\* `

-Destination $REPOSITORY\_ROOT/.github/ -Recurse -Force



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

# Containerize ứng dụng Java (backend)

## Phạm vi

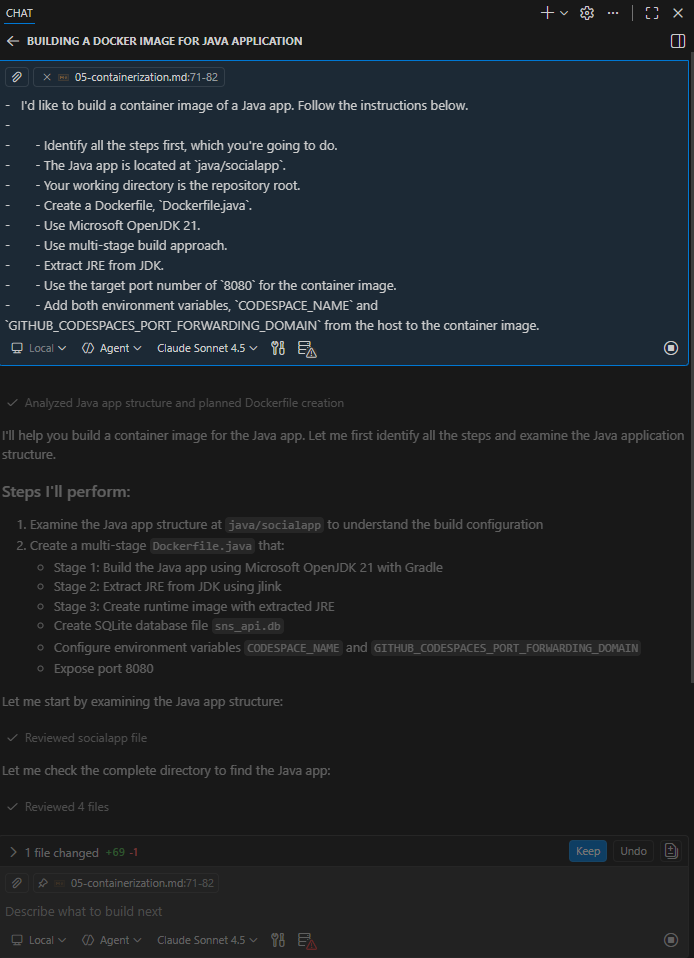
* + - Source của ứng dụng Java nằm tại java/socialapp
    - Tạo Dockerfile tên Dockerfile.java ở thư mục gốc repo

## Yêu cầu kỹ thuật

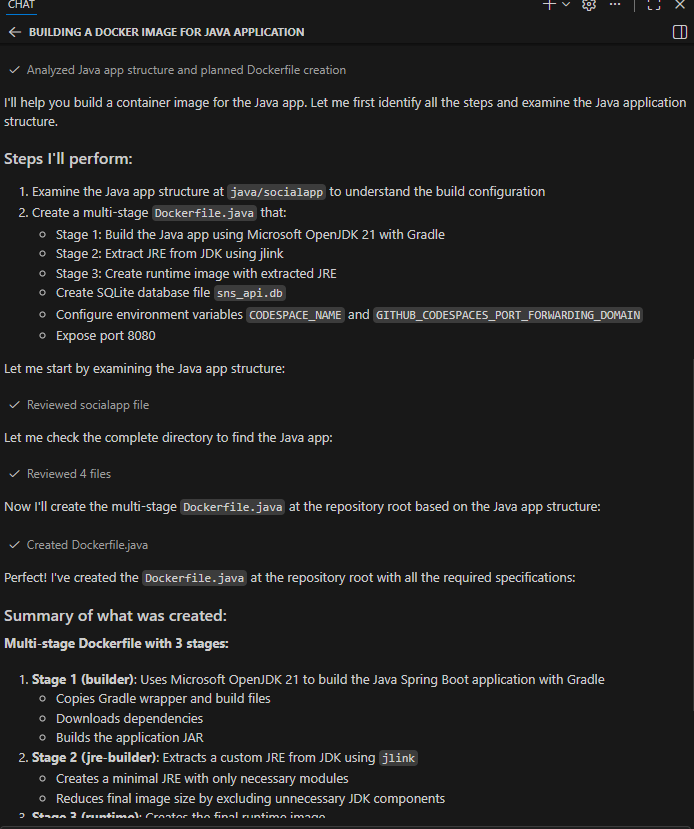
* + - Dùng Microsoft OpenJDK 21
    - Multi-stage build
    - Trích xuất JRE từ JDK (giảm dung lượng image runtime)
    - Expose/target port: 8080
    - Pass biến môi trường CODESPACE\_NAME và GITHUB\_CODESPACES\_PORT\_FORWARDING\_DOMAIN từ host vào container
    - Tạo file SQLite sns\_api.db trong image (không copy từ máy host)

## 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.



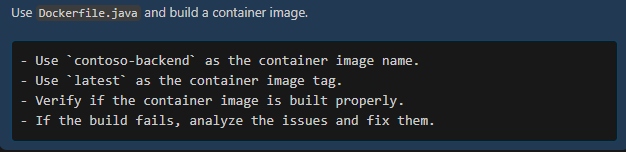
Hình 3. Prompt cho agent để đóng gói backend java



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

## Build image và kiểm tra

Prompt build image:



Hình 5. Prompt dùng dockerfile.java để build image

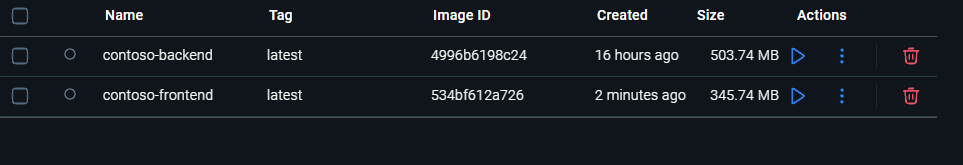
* Use `Dockerfile.java` and build a container image.
* - Use `contoso-backend` as the container image name.
* - Use `latest` as the container image tag.
* - Verify if the container image is built properly.
* - If the build fails, analyze the issues and fix them.



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

Kết quả kỳ vọng:

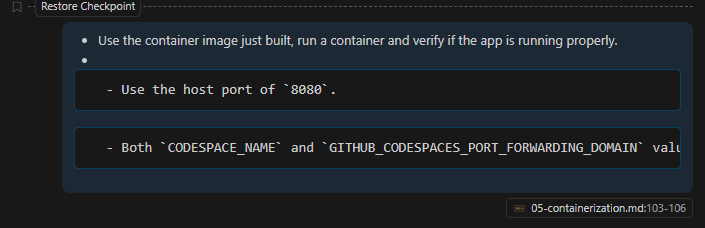
* + - Image contoso-backend:latest build thành công
    - Không còn lỗi Gradle/build
    - Container có thể khởi động và lắng nghe cổng 8080



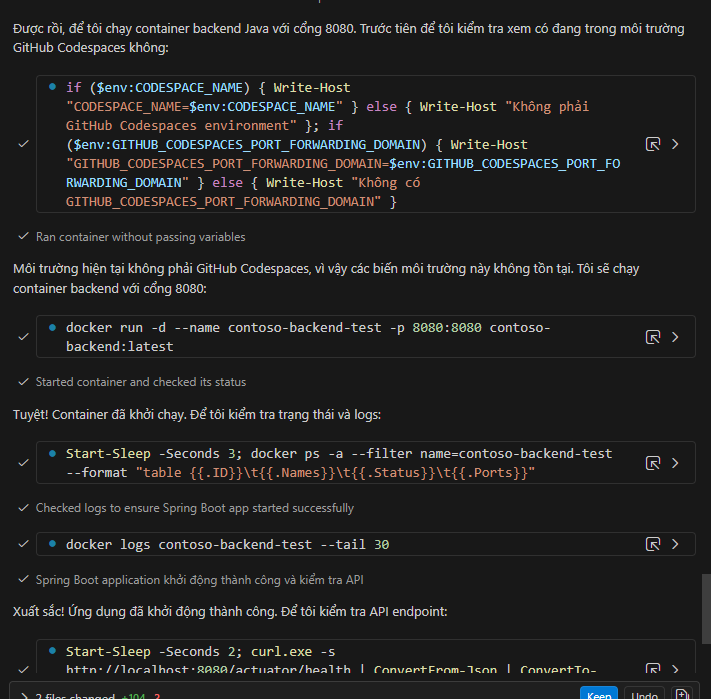
Hình 7. Các image đã được build thành công

## Chạy container và xác minh

Prompt chạy container:

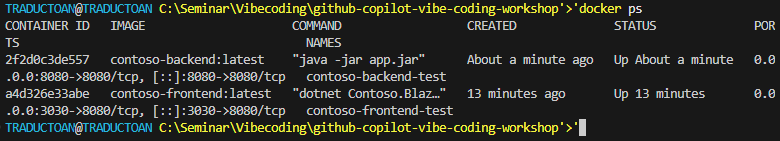


Hình 8. Prompt chạy container



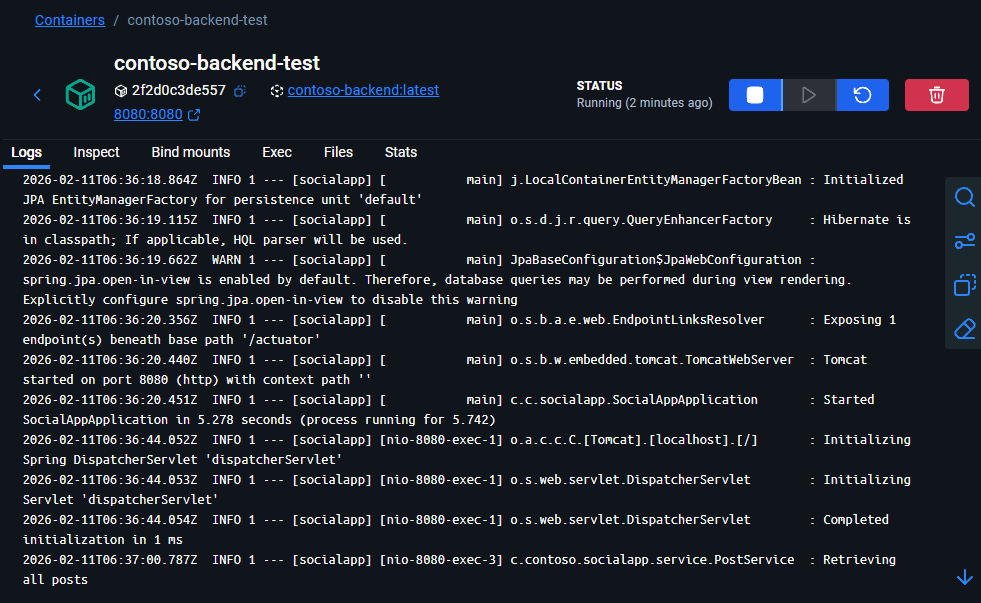
Hình 9. Phản hồi từ agent

* 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)



Hình 10. Docker ps

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



Hình 11. Log container backend

# Containerize ứng dụng .NET (frontend)

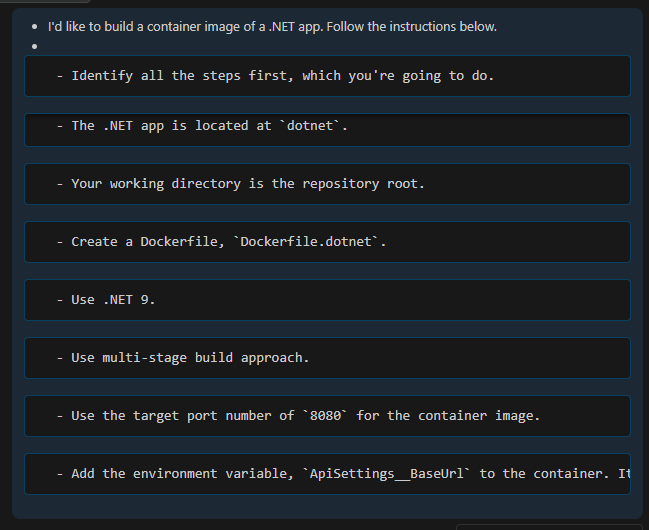
## Phạm vi

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

## Yêu cầu kỹ thuật

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

## 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`.

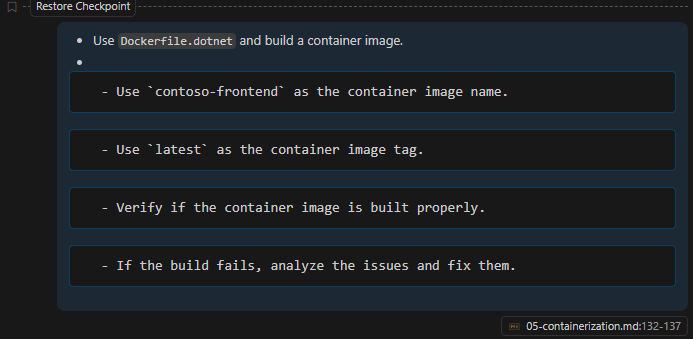
Hình 12. Prompt tạo dockerfile.dotnet

## Build image và kiểm tra

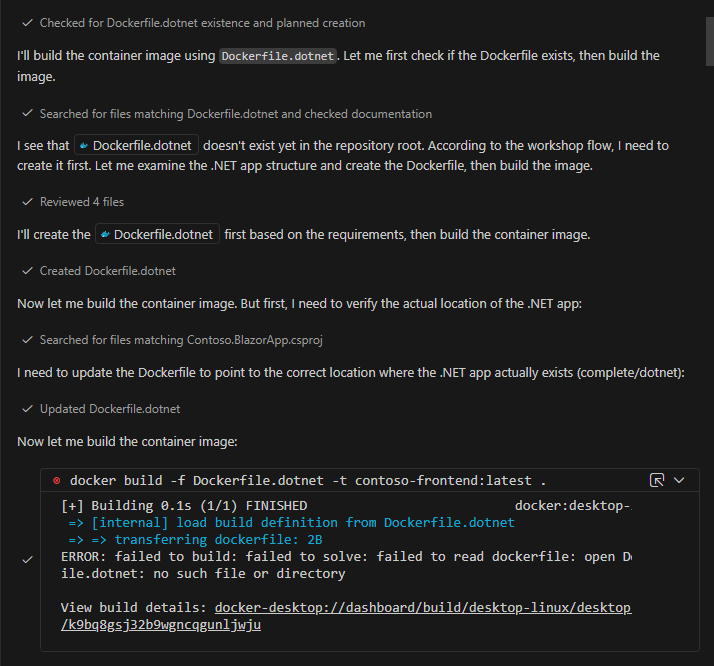
Prompt build image:

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

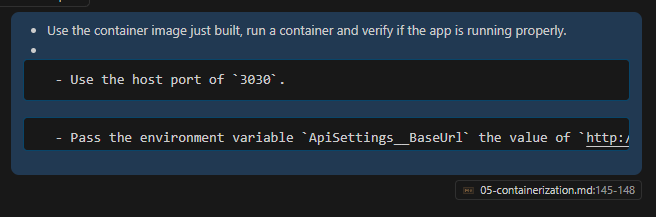


Hình 14. Phẩn hồi từ agent sau khi prompt build image

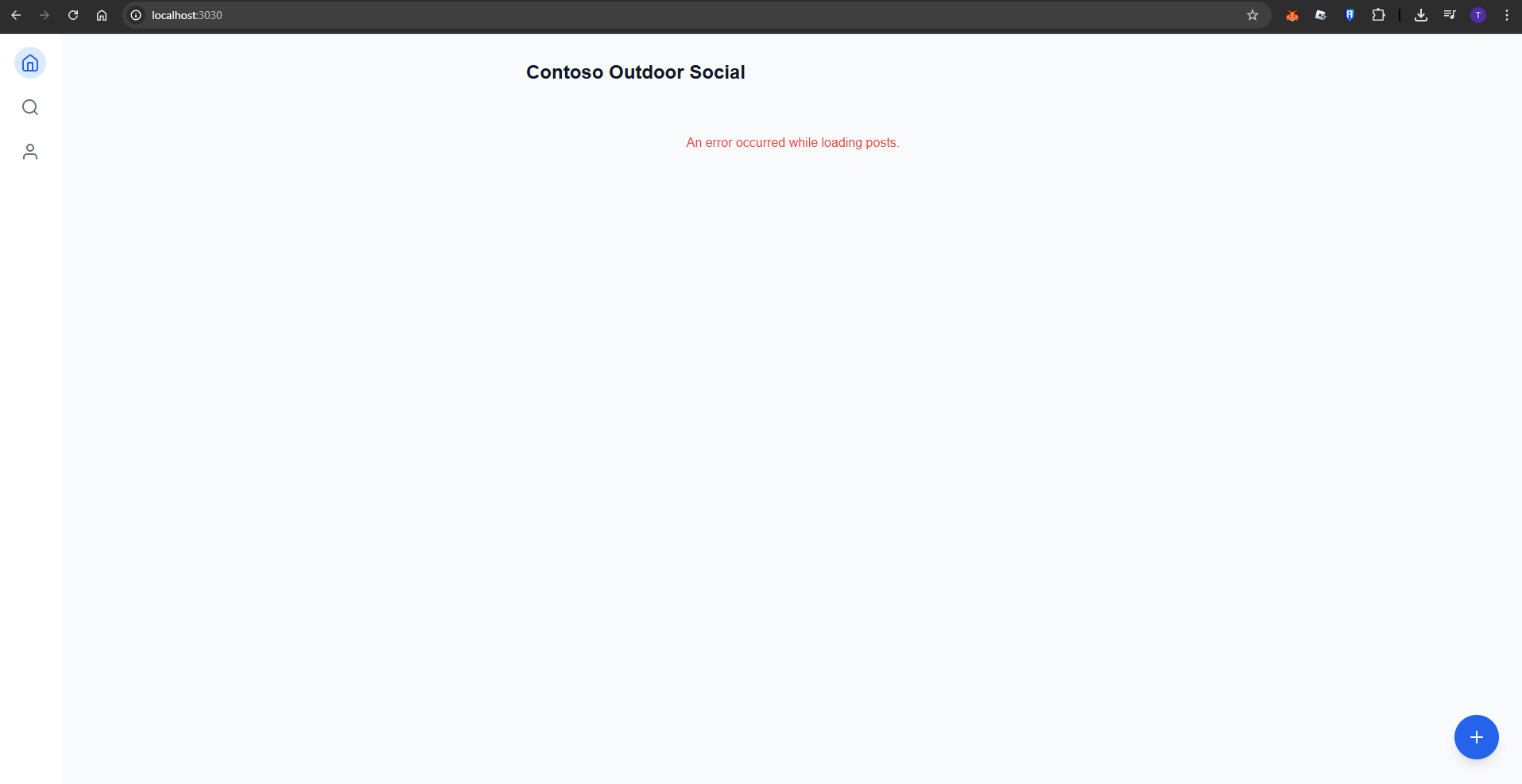
## Chạy container và xác minh

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 fontend dotnet



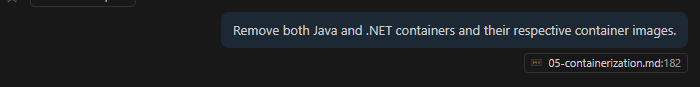
Hình 16. UI fontend

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.

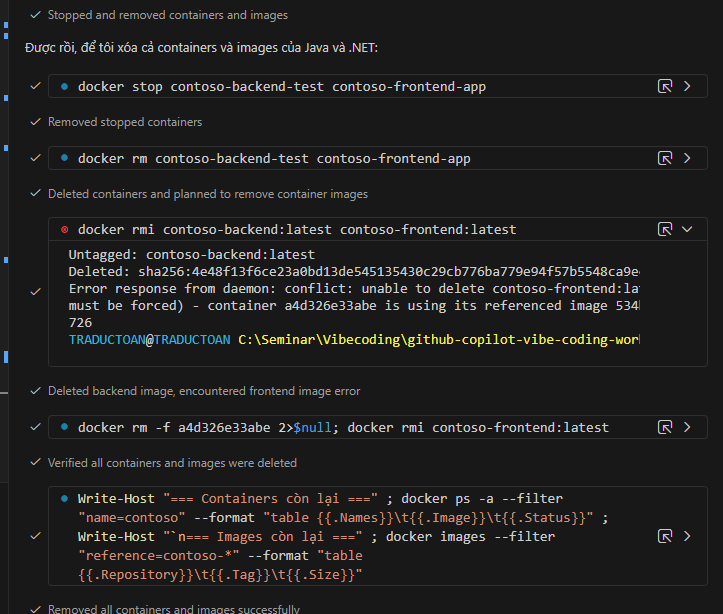
## Dọn dẹp tài nguyên

Mục đích: đảm bảo trạng thái sạch trước khi sang bước Docker Compose. Prompt tham khảo:

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



Hình 17. Prompt dọn dẹp tài nguyên



# Điều phối các container bằng Docker Compose

## Mục tiêu

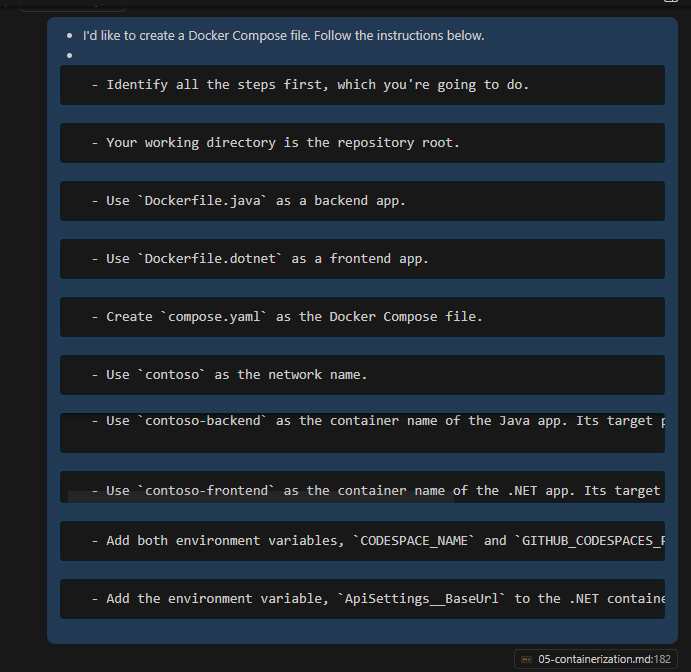
* + - Tạo compose.yaml để chạy cả backend và frontend cùng một network
    - Thiết lập biến môi trường và mapping cổng đúng yêu cầu

## Yêu cầu

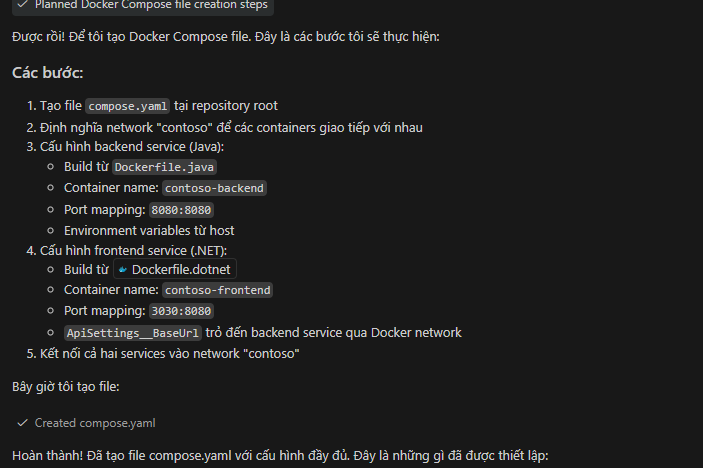
* + - Backend:
      * Dockerfile.java
      * container name: contoso-backend
      * ports: 8080:8080
      * network: contoso
      * pass CODESPACE\_NAME, GITHUB\_CODESPACES\_PORT\_FORWARDING\_DOMAIN
    - Frontend:
      * Dockerfile.dotnet
      * container name: contoso-frontend
      * ports: 3030:8080
      * network: contoso
      * ApiSettingsBaseUrl trỏ đến /api của backend

## 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, `ApiSettings\_\_BaseUrl` to the .NET container. It should point to the Java app's `/api`.

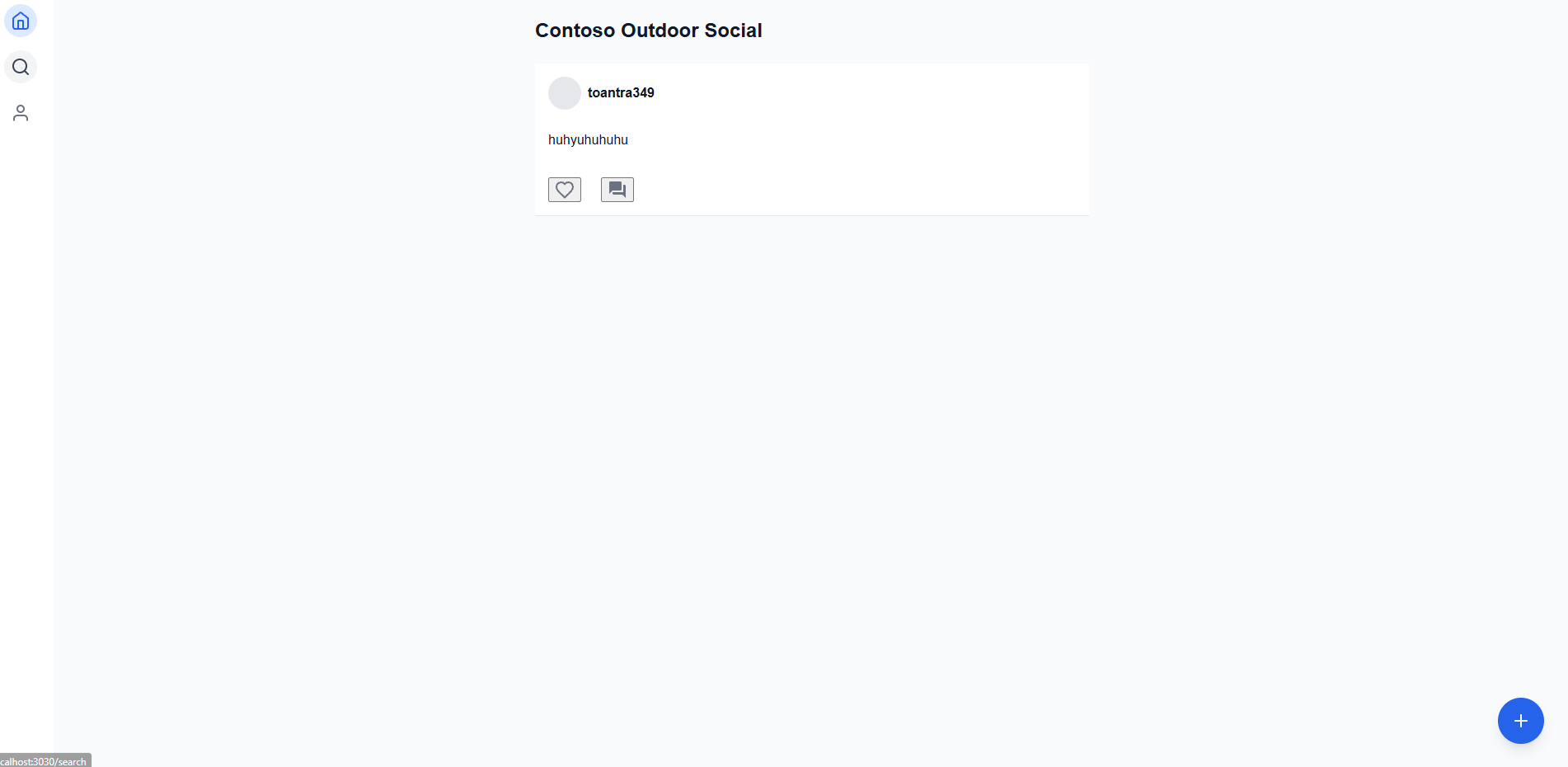


Hình 18. Prompt tạo docker compose



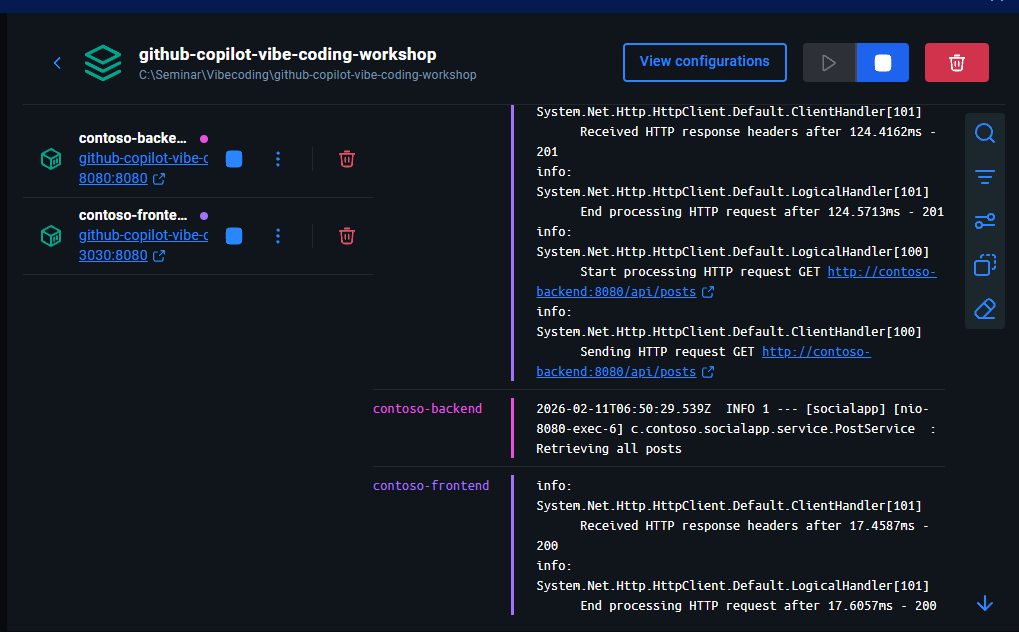
## Chạy compose và xác minh

Prompt chạy compose:



Hình 19. Web đã chạy hoàn chỉnh backend và fontend

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



Hình 20. Container sau khi docker compose

# 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](http://localhost:3030/)

# 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.