Điều khó khăn nhất khi làm **Microservice** là nhận dữ liệu để lưu trữ từ các service khác.

**Lợi**:

* Giúp truy vấn data nhanh chóng.
* Các service KO phụ thuộc vào nhau. Trường hợp 1 server die thì service hiện vẫn thực hiện truy vấn bình thường.

**Hại**:

* Trùng lặp dữ liệu.
* Có thể tốn chi phí bộ nhớ nhưng ko đáng kể.

Các service đều phải có **database riêng** và phải **độc lập với nhau**, **KO** dc kết nối trực tiếp với nhau để lấy dữ liệu.

Cách để lấy dữ liệu cần thiết giữa service khác với nhau ta sử dụng **Event-Bus** làm trung gian lắng nghe các events từ các service request lên.

Khi **Event-Bus** nhận được 1 **event** từ 1 **service** request lên thì **Event-Bus** sẽ **request** event đó lại cho tất cả các **service** dag hoạt động **kể cả service gửi event** để service nào cần thì sẽ nhận data đó về xử lý.

Có các Event-Bus khá mạnh (RabbitMQ, Kafka,...)

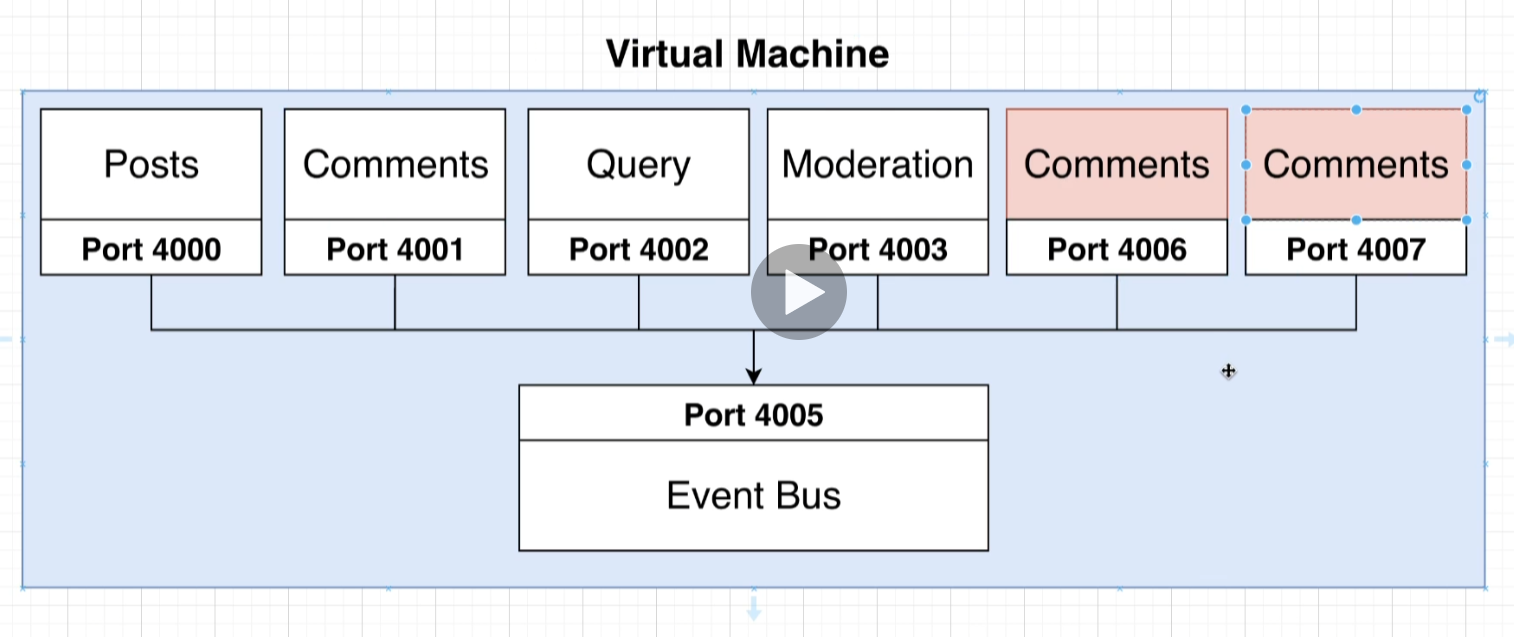
Khi 1 **service** cần xử lý request **bị die đột ngột**, sau khi **restart** lại thì **service** bị **mất request** đó và ko thể xử lý dc và request sẽ bị **pending** mãi mãi.

**Event-Bus Store** sẽ khắc phục tình trạng đó bằng cách **lưu toàn bộ dữ liệu** event do các service request lên. Khi restart lại service thì service đó sẽ tự động gọi request đến **Event-Bus Store** và lấy toàn bộ dữ liệu và request pending sẽ dc xử lý.

**DOCKER**:

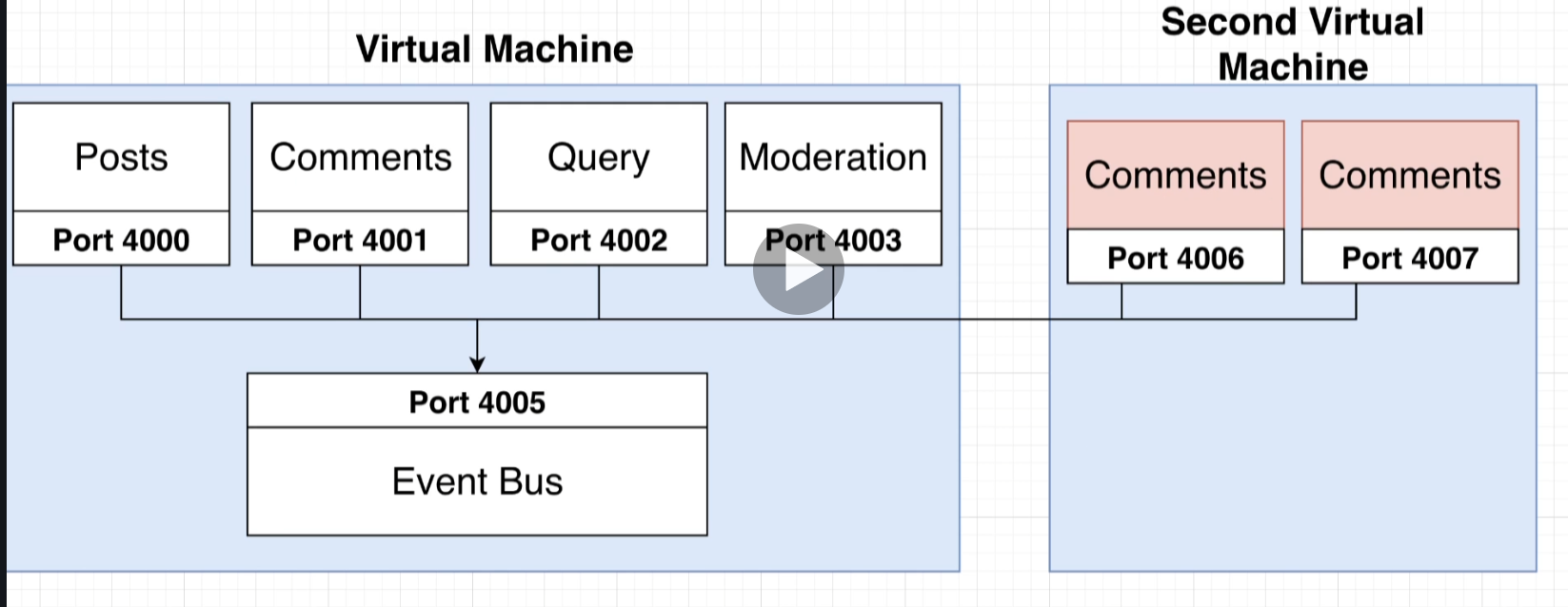
Khi ta deploy project **Microservice** lên server sẽ có các vấn đề xảy ra như sau:

* **Service** đó có lượng truy cập lớn và gây quá tải, xử lý điều đó ta phải **Clone** ra thêm **Service** đó để khi user truy cập tới ta có thể **Load Balance** (đẩy request vào 1 trong các Service instance bất kỳ)



**NHƯNG** điều này sẽ gây ra vấn đề, ta phải vào source của **Event-Bus** và config thêm 2 request đến 2 **PORT** của các **Clone-Services** và deploy lại. **(Điều này quá bất tiện)**

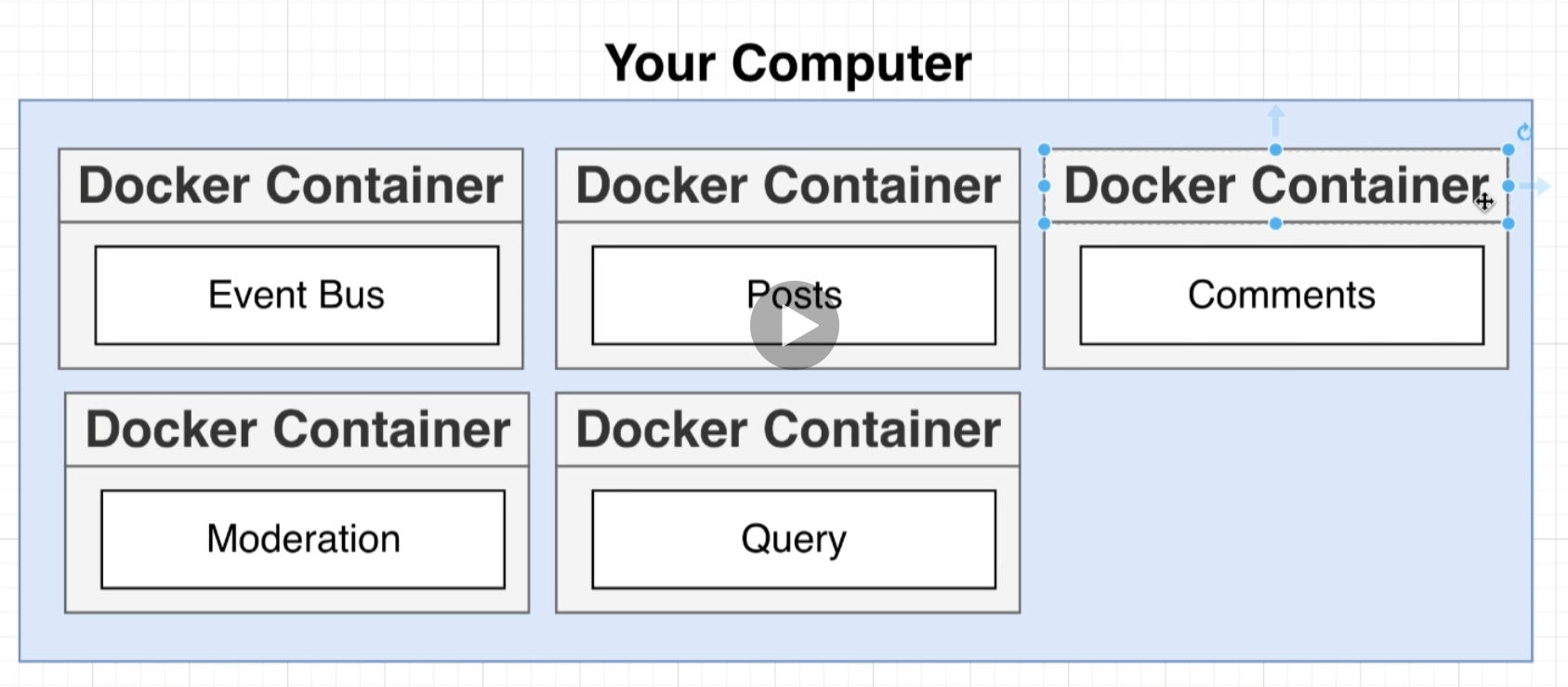
* Khi **Clone** nhiều **Service** như vậy sẽ dẫn đến việc **Server** **KO** đủ tài nguyên nên ta sẽ phải thuê thêm 1 **Server** khác để chứa các **Clone-Services**



Trường hợp này cũng sẽ gây ra vấn đề như trên.

Giả sử như WEB của mình có lượng truy cập lớn vào 10h sáng mỗi ngày nên mình muốn 10h sáng sẽ có 2 Clone-Service dc active để xử lý request. Nhưng vào lúc 1h khuya mình chỉ muốn 1 Service chạy chính và shutdown Server của Clone-Service để tiết kiệm chi phí.

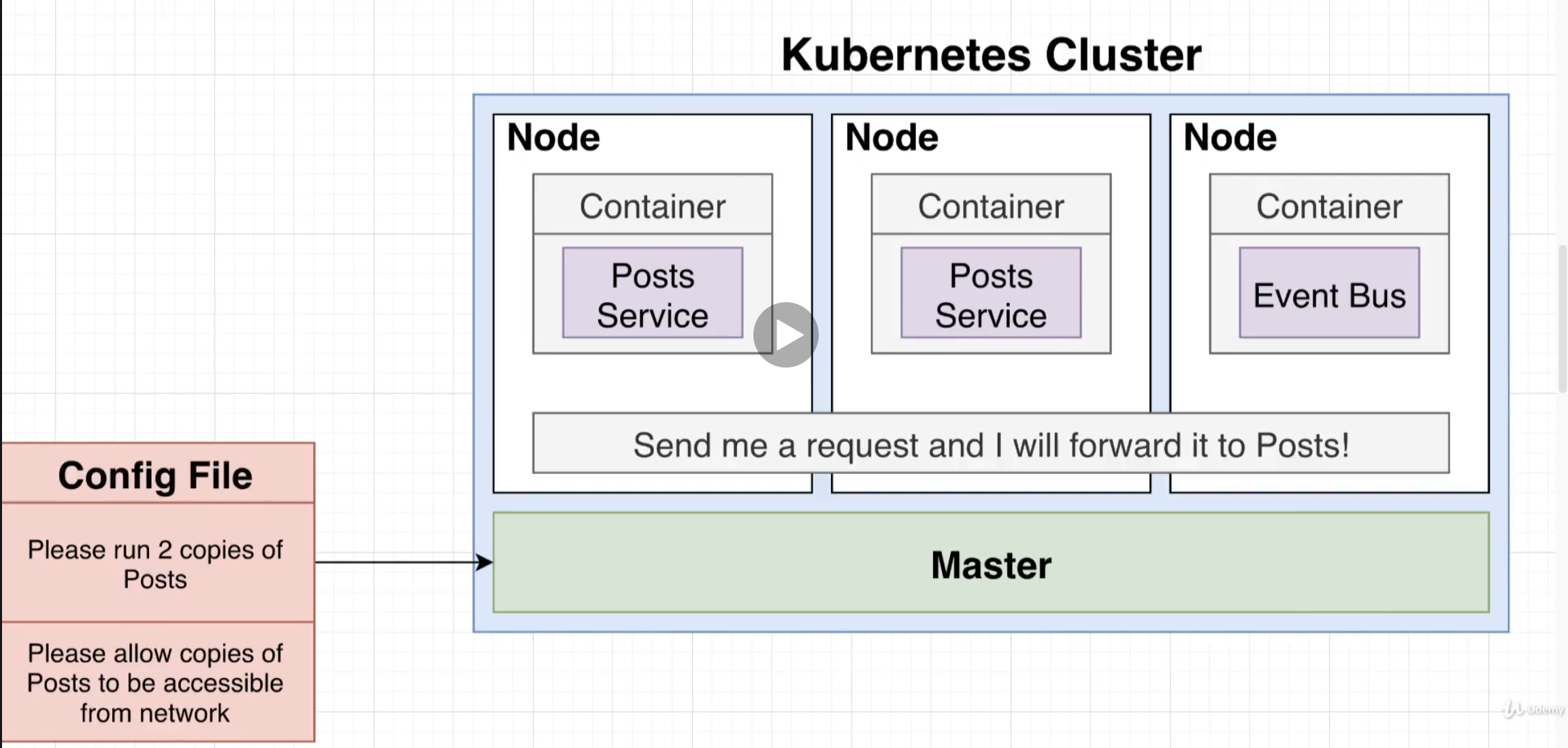
**DOCKER & KUBERNETES** sẽ giúp giải quyết vấn đề đó.



**Container** là môi trường máy tính độc lập. **Container** chứa những thứ cần thiết để chạy 1 chương trình.

Ta sẽ tạo các Container tách biệt. Mỗi Container sẽ chứa 1 Service.

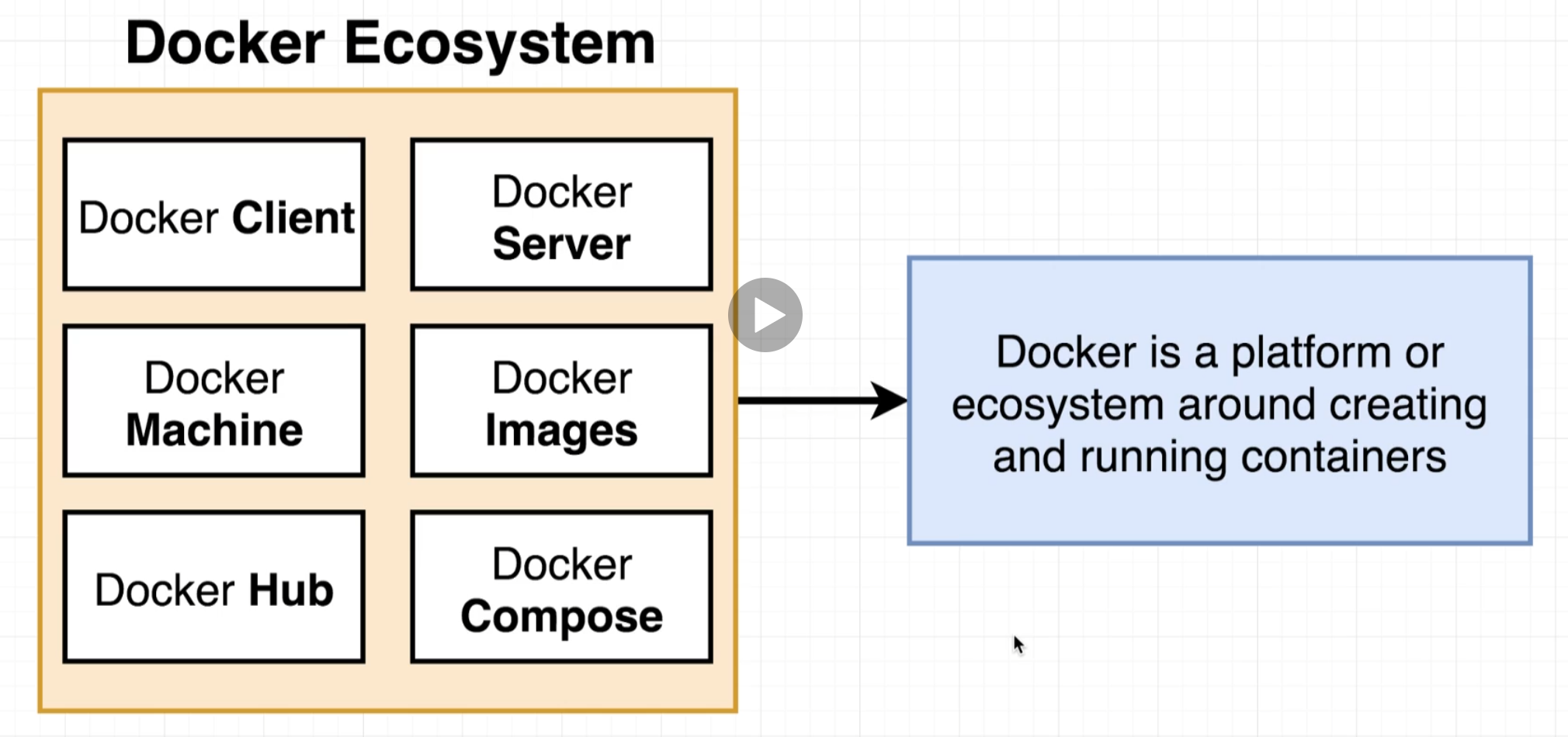
Khi muốn Clone 1 Service ta sẽ tạo 1 **Docker Container** mới.

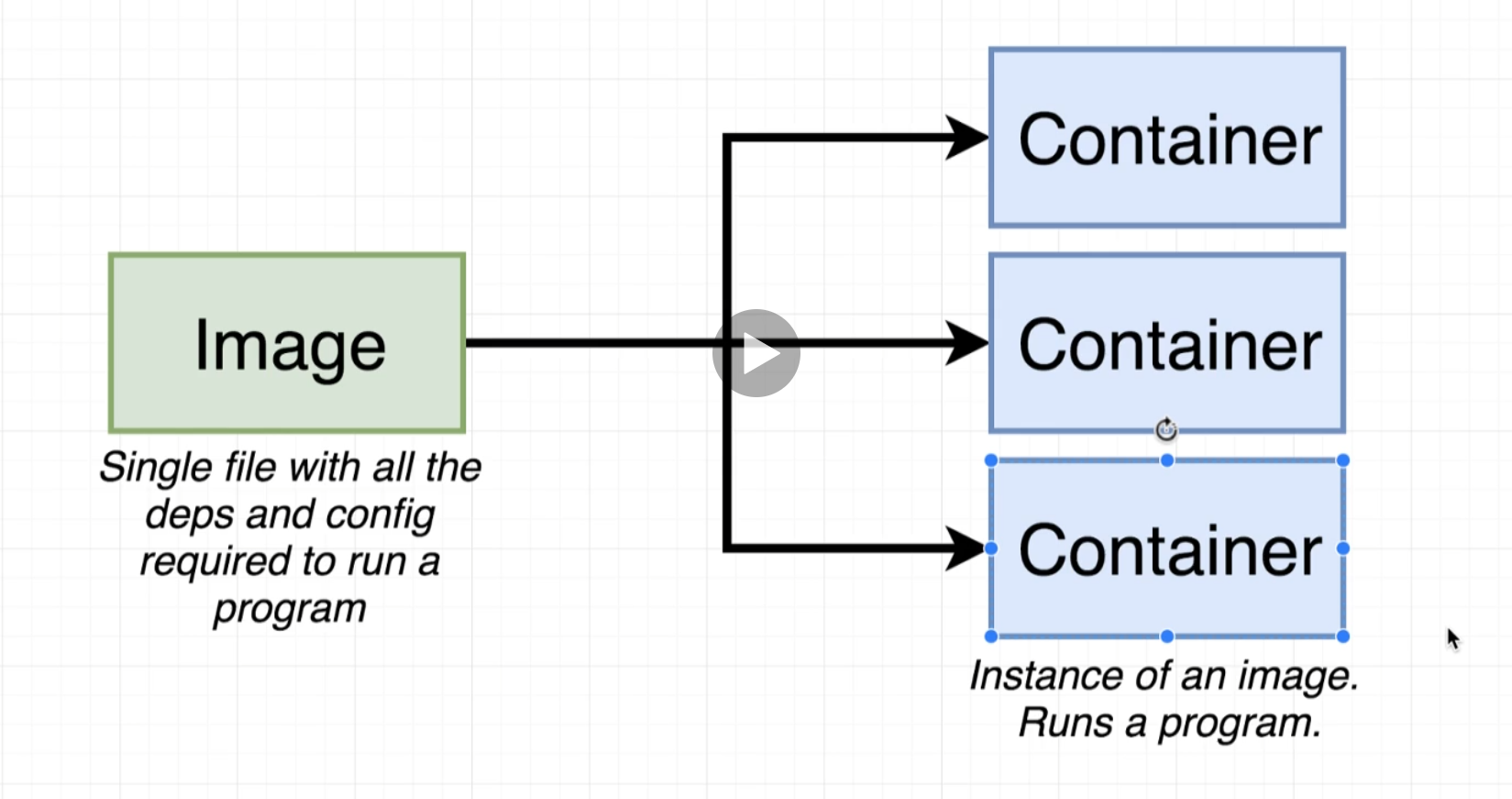


**Kubernetes** có thể giúp ta có khả năng **giao tiếp** giữa các **Service** với nhau thông qua **1 Third-Party**.

Nếu **KO** có **Kubernetes** ta phải xử lý **Event-Bus** để kết nối tới từng **Service**.







Command: **docker run <name> <override command>**

**~~ [docker create <name> <override command> + docker start <–a> <container\_id>]**

* Ta muốn **tạo** ra 1 **Container** sử dụng **Image** tên **\*name\*.**
* **Override command** sẽ được thực thi bên trong **Container**.

Command: **docker ps**

* Xem các **Container** dag chạy.

Command: **docker ps –all**

* Xem toàn bộ các **Container** hiện hành ở máy tính.

Command: **docker system prune**

* Xóa toàn bộ các **Stopped Containers** bao gồm **Image Cache**.

Command: **docker logs <container id>**

* Xem log của container đó.

Command: **docker stop/kill <container id>**

* Dừng container đó.
* Stop: dừng sau 1 khoảng tgian ngắn để thực hiện việc dọn dẹp, save file**.(Nên dùng)**
* Kill: dừng ngay lập tức.

Command: **docker exec –it <container id> <command>**

* Cho phép thực thi command bên trong container đó.