Integration of Docker Applications

Backend Services

Table of Contents

[Introduction 2](#_Toc499133443)

[Docker terminology 2](#_Toc499133444)

[Integration of Docker Backend Services 3](#_Toc499133445)

[Direct client-to-microservice communication 3](#_Toc499133446)

[Using an API Gateway 3](#_Toc499133447)

[NGINX Reverse Proxy API Gateway Setup 4](#_Toc499133448)

[Other tools for Integration 6](#_Toc499133449)

[High Performance and Scalable tool 6](#_Toc499133450)

[Reactive Programming Model tool 6](#_Toc499133451)

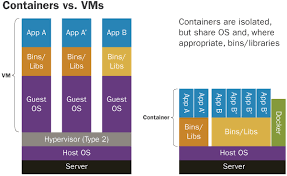
[Service Invocation tool 6](#_Toc499133452)

[Azure API Management 6](#_Toc499133453)

[Summary 7](#_Toc499133454)

# Introduction

Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises. Docker image, containers, run natively on Linux and Windows. Windows images run only on Windows hosts and Linux images run only on Linux hosts. The host is a server or a VM. Containerization is an approach in which an application or service, its dependencies, and its configuration are packaged together as a container image. Containers offer the benefits of isolation, portability, agility, scalability, and control. The most important benefit of container is the isolation provided between Dev and Ops. Scaling can be done quickly by creating new containers for short-term tasks. Reliability can be managed by running multiple instances of the same container image across multiple host servers. Thus, it saves cost, solves deployment problem, and improves DevOps and production operations. The difference between container, docker and VM is presented in figure below:



Virtual machines include the application, the required libraries or binaries, and a full guest operating system. Containers include the application and all its dependencies. However, share the OS kernel with other containers.

# Docker terminology

**Container image**: A image includes all dependencies plus deployment and execution configuration to be used by container runtime.

**Tag**: A mark or label you can apply to images so that different images or versions of the same image can be identified.

**Dockerfile**: A text file that contains instructions for how to build a Docker image.

**Build**: The action of building a container image based on the information and context provided by its Dockerfile.

**Repository (repo)**: A collection of related Docker images, labeled with a tag that indicates the image version. Some repos contain multiple variants of a specific image, such as an image containing SDKs (heavier), an image containing only runtimes (lighter), etc. Some repo can contain platform variants, such as a Linux image and a Windows image.

**Registry**: A service that provides access to repositories. The default registry for most public images is [Docker Hub](https://hub.docker.com/) (owned by Docker as an organization

**Docker Hub**: Docker Hub is a public registry that provides Docker image hosting, public or private registries, build triggers and web hooks, and integration with GitHub and Bitbucket.

**Azure Container Registry**: This provides a registry that is close to your deployments in Azure and that gives you control over access, making it possible to use your Azure Active Directory groups and permissions.

**Docker Trusted Registry (DTR)**: A Docker registry service (from Docker) that can be installed on-premises so it lives within the organization’s datacenter and network.

**Docker Community Edition (CE)**: Development tools for Windows and macOS for building, running, and testing containers locally both for Linux and Windows.

**Compose**: A command-line tool and YAML file format with metadata for defining and running multi-container applications.

**Cluster**: A collection of Docker hosts exposed as if it were a single virtual Docker host, so that the application can scale to multiple instances of the services spread across multiple hosts within the cluster. Docker clusters can be created with Docker Swarm, Mesosphere DC/OS, Kubernetes, and Azure Service Fabric.

**Orchestrator**: An orchestrator is responsible for running, distributing, scaling, and healing workloads across a collection of nodes. Orchestrators enable you to manage their images, containers, and hosts through a command line interface (CLI) or a graphical UI.

# Integration of Docker Backend Services

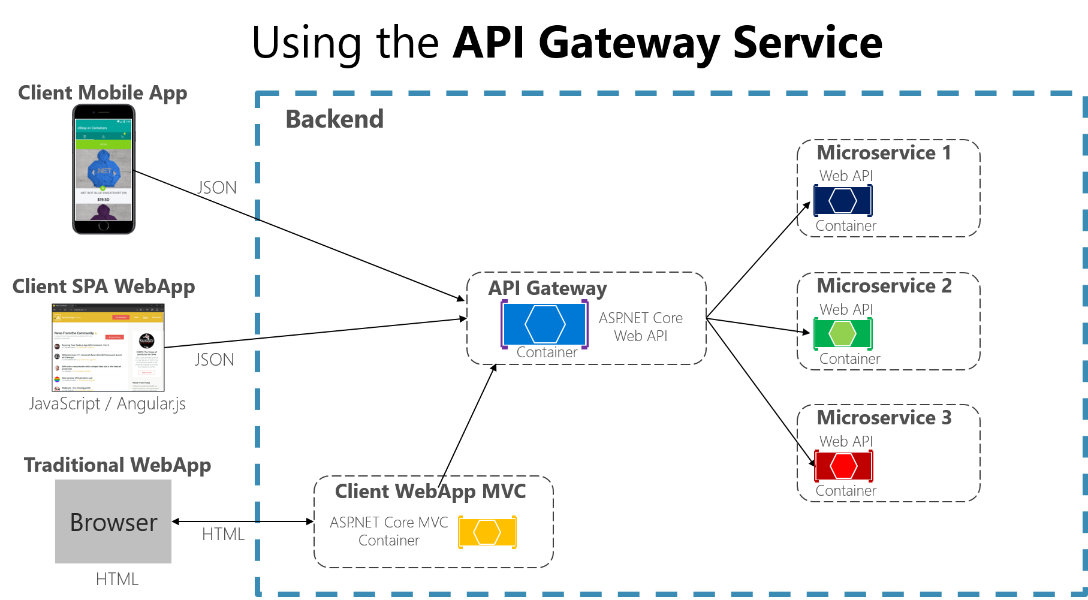
A microservices architecture is an approach to building a server application as a set of small backend services. Each service runs in its own process and communicates with other processes using protocols such as HTTP/HTTPS, WebSockets, or AMQP. Ways of communication between client and backend services are:

## Direct client-to-microservice communication

In this approach, a client app can make requests directly to the microservices and each microservice has a public endpoint. A direct client-to-microservice communication architecture could be good for a small microservice-based application, especially if the client app is a server-side web application like an ASP.NET MVC app. However, when you build large and complex microservice-based applications and when the client apps are remote mobile apps or SPA web applications, that approach faces a few issues like security, latency, complexity etc.

## Using an API Gateway

API Gateway is good approach when design is large or complex microservice-based applications with multiple client apps. This is a service that provides a single-entry point for certain groups of microservices. The API Gateway is responsible for request routing, composition, and protocol translation. It provides each of the application’s clients with a custom API. The API Gateway can also mask failures in the backend services by returning cached or default data. Example is NGINX API gateway.



But it isn't a good idea to have a single API Gateway aggregating all the internal microservices of your application. If it does, it acts as a monolithic aggregator or orchestrator and violates microservice autonomy by coupling all the microservices. Therefore, the API Gateways should be segregated based on business boundaries and not act as an aggregator for the whole application

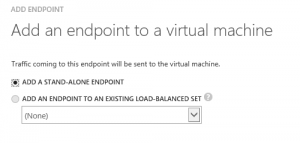
### NGINX Reverse Proxy API Gateway Setup

A reverse proxy is a way to expose an internal webserver to the outside world. Sites and services using those hostnames like “localhost” or “127.0.0.1” are not accessible from other computers on the network. To setup a reverse proxy on Windows Azure using Nginx, following is required:

* Azure virtual machine running Windows Server
* Download Nginx for Windows from <http://nginx.org/en/download.html>.
* A website running locally with URL as http://127.0.0.1:11235

Step 1: Open the port in Azure Portal

* Log in to your Azure portal and select your virtual machine
* Add an endpoint to Virtual machine by selecting radio button of “Add a standalone endpoint” option.

[](https://i1.wp.com/wbsimms.wpengine.com/wp-content/uploads/2015/01/EndpointsAdd.png)

* Next, add your endpoint details
  + Name
  + Protocol: TCP
  + Public Port: external port you’ll connect to
    - * e.g. http://<Your VM Name>.cloudapp.net:<External Port>
  + Private Port: For most cases, this will be same as the public port

Step 2: Setup Nginx

* Copy the Nginx zip file to the VM
* Unzip the file
* Go to the “conf” folder
* Create a new file named “proxy.conf” with the following content:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | proxy\_redirect          off;  #proxy\_set\_header        Host            $host;  proxy\_set\_header        X-Real-IP       $remote\_addr;  proxy\_set\_header        X-Forwarded-For $proxy\_add\_x\_forwarded\_for;  client\_max\_body\_size    10m;  client\_body\_buffer\_size 128k;  proxy\_connect\_timeout   90;  proxy\_send\_timeout      90;  proxy\_read\_timeout      90;  proxy\_buffers           32 4k; |

* + line #2 can be removed in case some internal servers chokes
* Update the nginx.conf  with the following
  + server {  
    listen       81;  
    server\_name <You VM Name>.cloudapp.net localhost;location / {proxy\_pass http://127.0.0.1:11235/;  
    include proxy.conf;  
    }}
* This instructs Nginx to listen on port 81 and serve up content from 127.0.0.1:11235

Step 3: Start and test the proxy

* Run nginx.exe from the command-line
* Open a browser and navigate to http:<You VM Name>.cloudapp.net:<External Port>
* You should see your local content published but with the new hostname and port
* Check the Nginx logs if you have any errors.

# Other tools for Integration

## High Performance and Scalable tool

To build the API Gateway on a platform that supports asynchronous, nonblocking I/O for high performance and scalability, some tools are:

* NGINX Plus: [NGINX Plus](http://www.nginx.com/solutions/api-gateway/) offers a mature, scalable, high‑performance web server and reverse proxy that is easily deployed, configured, and programmed. NGINX Plus can manage authentication, access control, load balancing requests, caching responses, and provides application‑aware health checks and monitoring
* Netty, Vertx, Spring Reactor, or JBoss Undertow: On the Java Virtual Machine.

## Reactive Programming Model tool

To minimize response time, the API Gateway should perform independent requests concurrently. The API Gateway might first need to validate the request by calling an authentication service, before routing the request to a backend service. Tools for Asynchronous, Event-Driven approach are:

* Future in Scala
* CompletableFuture in Java 8,
* Promise in JavaScript.
* Reactive Extensions (also called Rx or ReactiveX), originally developed by Microsoft for the .NET platform
* Netflix created RxJava for the.
* RxJS for JavaScript, which runs in both the browser and Node.js.

## Service Invocation tool

There are two styles of inter‑process communication. One option is to use an asynchronous, messaging‑based mechanism that supports standard protocols such as AMQP and STOMP. Tools for asynchronous, messaging‑based mechanism are:

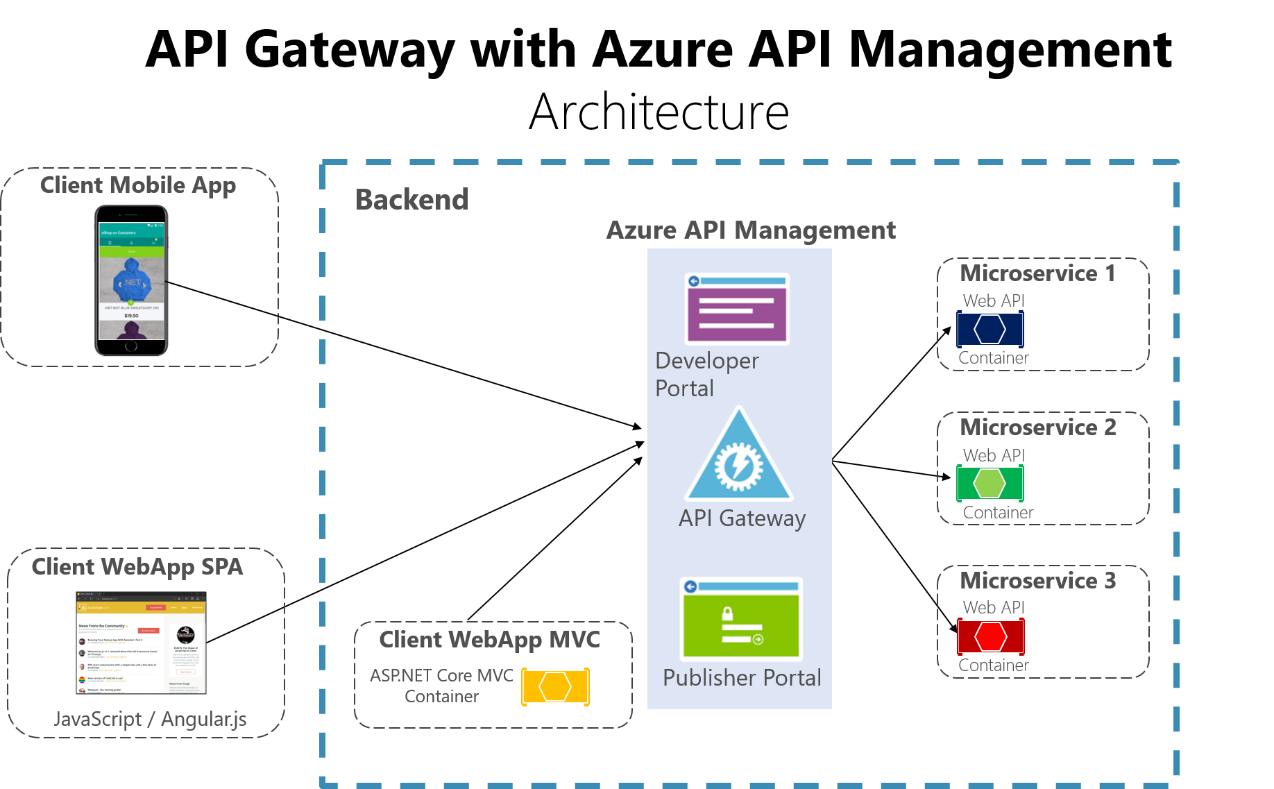
* RabbitMQ
* Apache Kafka
* Apache ActiveMQ
* NSQ
* Zeromq

The other style of inter‑process communication is a synchronous mechanism such as HTTP. Tools are:

* REST
* Apache Thrift

## Azure API Management

Azure API management not only solves your API Gateway needs, but provides features like gathering insights from your APIs. If you are using an API management solution, an API Gateway is only a component within that full API management solution.



This acts more like a reverse proxy for ingress communication, where you can also filter the APIs from the internal microservices plus apply authorization to the published APIs in this single tier. The insights available from an API Management system help you get an understanding of how your APIs are being used and how they are performing. They do this by letting you view near real-time analytics reports and identifying trends that might impact your business. Plus, you can have logs about request and response activity for further online and offline analysis.

With Azure API Management, you can secure your APIs using a key, a token, and IP filtering. These features let you enforce flexible and fine-grained quotas and rate limits, modify the shape and behavior of your APIs using policies, and improve performance with response caching.

# Summary

With a monolithic application, there is just one set of endpoints. In a microservices architecture, however, each microservice exposes a set of fine‑grained endpoints. Thus, it makes sense to implement an API Gateway, which acts as a single-entry point into a system. The API Gateway is responsible for request routing, composition, and protocol translation.