Let us suppose we are building a cloud native application. [4]

We have a [5] web application as well as a [6] mobile application, and we are going to use a microservice architecture hosted in the [10] cloud.

[12] In a microservices architecture, each microservice

* Represents a bounded context such as [19] promotions, [20] deliveries, or [21] authentication
* And exposes a set of [24] RESTful APIs which provide communications with the client applications.

[30]

In our example the client will [32] authenticate (or log in) directly with the authentication service, which returns a token. [35] This token is then passed onto each of the other services such as promotions, which in turn validates that token with the authentication service.

[40] Microservices are added over time, for example a reporting service is added require changes in both client applications.

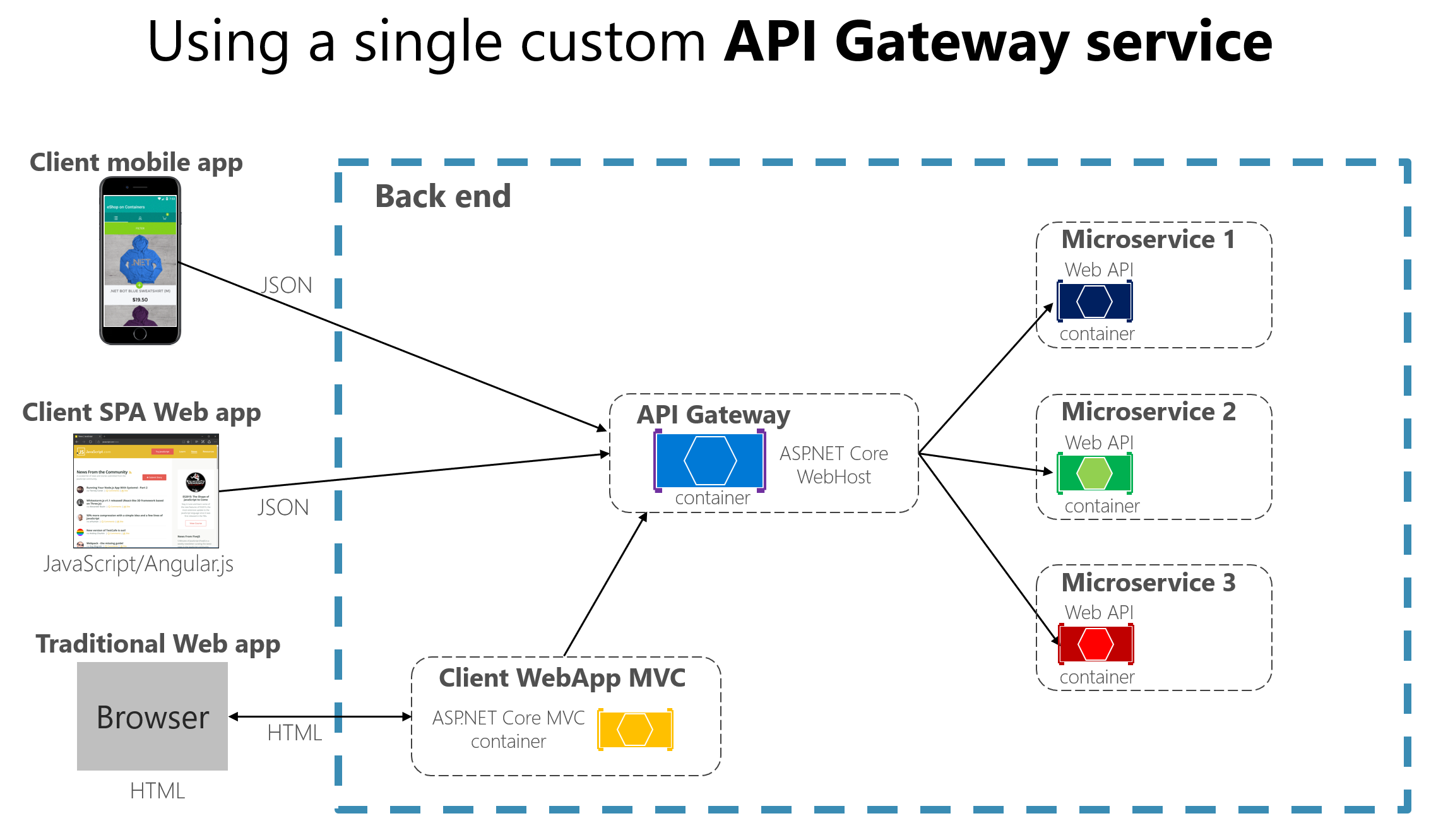
Because the client applications need to know where to find the services, each microservice requires a unique [55] DNS name.

## Drivers

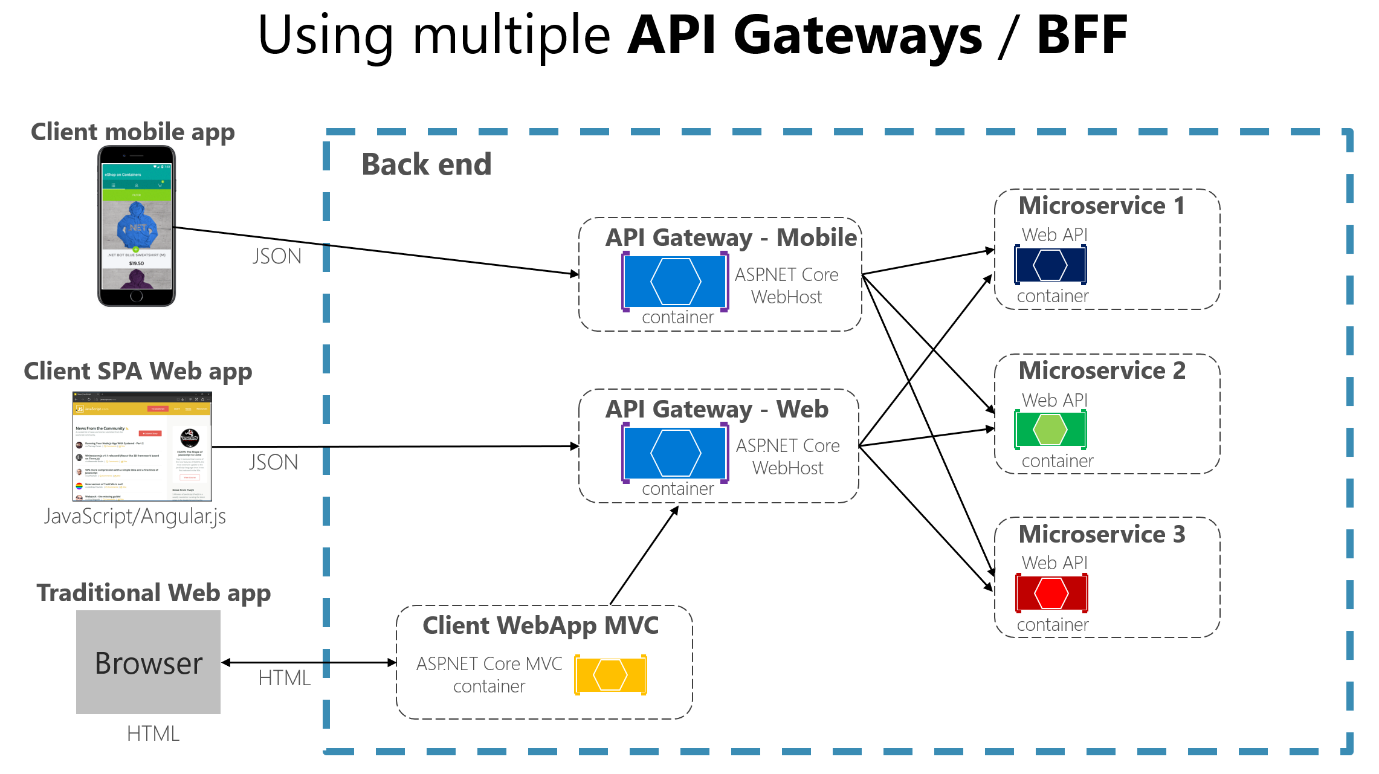
* Different clients need different data. For example, the desktop browser version of a product details page desktop is typically more elaborate then the mobile version.
* Network performance is different for different types of clients. For example, a mobile network is typically much slower and has much higher latency than a non-mobile network. And, of course, any WAN is much slower than a LAN. This means that a native mobile client uses a network that has very difference performance characteristics than a LAN used by a server-side web application. The server-side web application can make multiple requests to backend services without impacting the user experience where as a mobile client can only make a few.
* The number of service instances and their locations (host+port) changes dynamically
* Partitioning into services can change over time and should be hidden from clients
* Services might use a diverse set of protocols, some of which might not be web friendly
* The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. For example, as described above, a client needing the details for a product needs to fetch data from numerous services.

## Solution

An API Gateway acts as a single-entry point for RESTful calls to the backend.



Also known as a “backend for frontend” (BFF).



## Main Features

**Reverse proxy** – redirects request (layer 7 routing) of HTTP requests to the endpoints of the internal microservices. The client has no view of the internals of the backend, and therefore the backend is free to refactor services. This is a big benefit when evolving a monolith toward microservices.

**Cross-cutting concerns or gateway offloading** –provide cross-cutting features such as

* Authentication and authorization
* Service discovery integration
* Response caching
* Retry policies, circuit breaker, and QoS
* Rate limiting and throttling
* Load balancing
* Logging, tracing, correlation
* Headers, query strings, and claims transformation
* IP whitelisting
* SSL termination
* Web application firewall
* GZIP compression
* Servicing static content

## Drawbacks

**Coupling** – each of the microservice API’s are now coupled to the gateway. This may cause difficulties if the contract for the microservices is ever changing.

**Another Moving Part** – This adds yet another part to the application which creates an additional point of failure.

**Latency** – Increases response time to the client. However, this is small and only becomes an issue when there is a requirement for exceptionally low latency between the front and back ends.

**Additional Development and Maintenance** – Regardless of the product chosen, addition development and maintenance time is required. This development effort can bottleneck the release of the microservices.