**Why consider API Gateways instead of direct client-to-microservice communication**

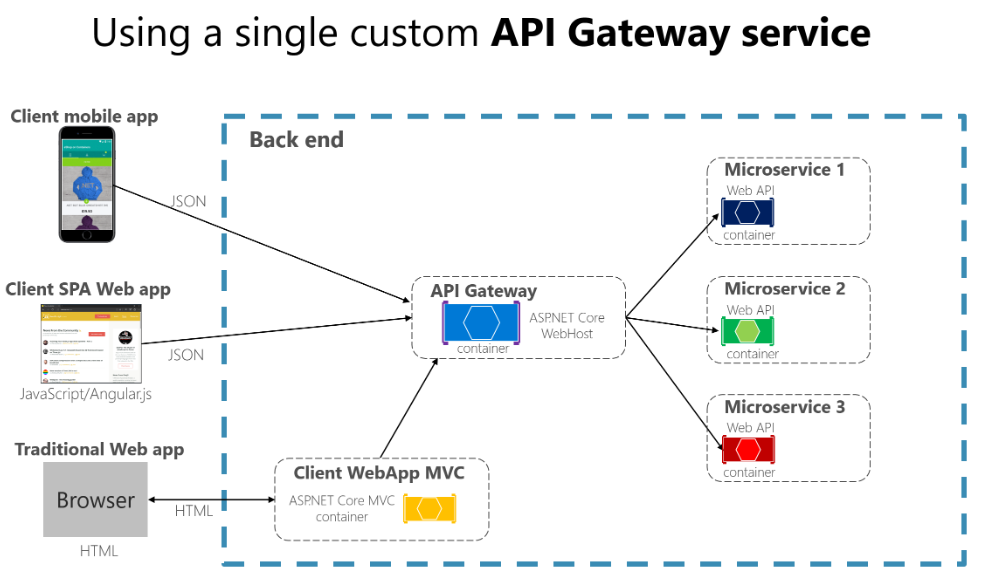
In a microservices architecture, the client apps usually need to consume functionality from more than one microservice. If that consumption is performed directly, the client needs to handle multiple calls to microservice endpoints. What happens when the application evolves and new microservices are introduced or existing microservices are updated? If your application has many microservices, handling so many endpoints from the client apps can be a nightmare. Since the client app would be coupled to those internal endpoints, evolving the microservices in the future can cause high impact for the client apps.

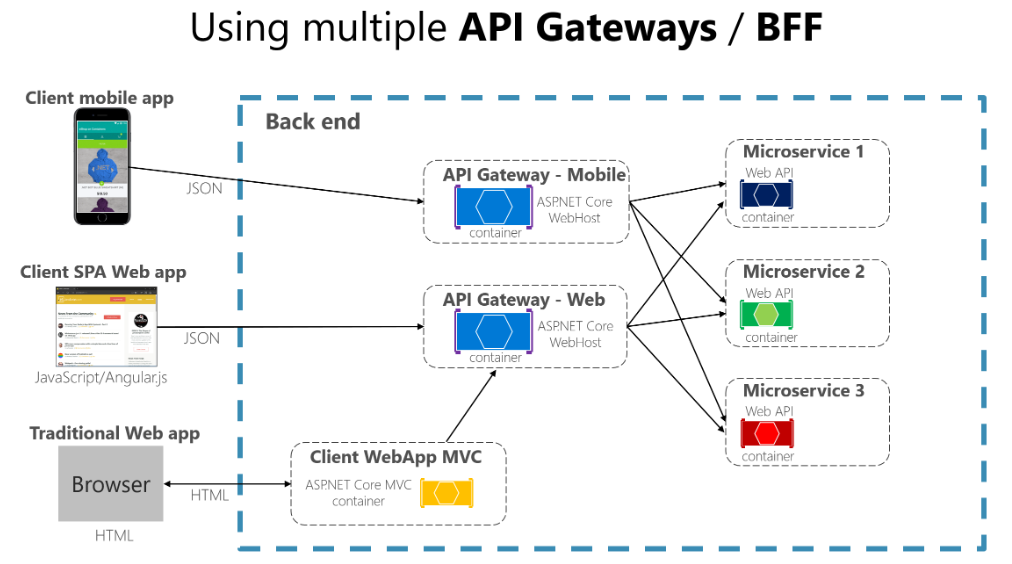
Therefore, having an intermediate level or tier of indirection (Gateway) can be very convenient for microservice-based applications. If you don't have API Gateways, the client apps must send requests directly to the microservices and that raises problems, such as the following issues:

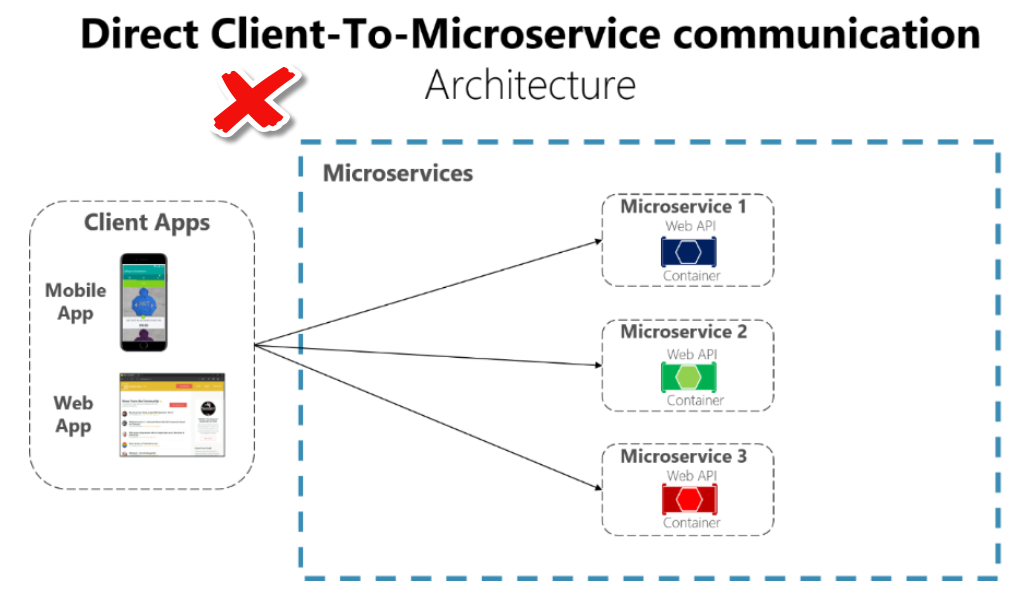
* **Coupling**: Without the API Gateway pattern, the client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance pretty badly because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated frequently, making the solution harder to evolve.
* **Too many round trips**: A single page/screen in the client app might require several calls to multiple services. That can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
* **Security issues**: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that aren't directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
* **Cross-cutting concerns**: Each publicly published microservice must handle concerns such as authorization, SSL, etc. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

## What is the API Gateway pattern?

The API gateway sits between the client apps and the microservices. It acts as a reverse proxy, routing requests from clients to services. It can also provide additional cross-cutting features such as authentication, SSL termination, and cache.







## Main features in the API Gateway pattern

**Reverse proxy or gateway routing**

**Requests aggregation**

**Cross-cutting concerns or gateway offloading**

* 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

## Drawbacks of the API Gateway pattern

* The most important drawback is that when you implement an API Gateway, you're coupling that tier with the internal microservices
* Using a microservices API Gateway creates an additional possible single point of failure.
* An API Gateway can introduce increased response time due to the additional network call. However, this extra call usually has less impact than having a client interface that's too chatty directly calling the internal microservices.
* If not scaled out properly, the API Gateway can become a bottleneck.
* An API Gateway requires additional development cost and future maintenance if it includes custom logic and data aggregation. Developers must update the API Gateway in order to expose each microservice's endpoints. Moreover, implementation changes in the internal microservices might cause code changes at the API Gateway level. However, if the API Gateway is just applying security, logging, and versioning (as when using Azure API Management), this additional development cost might not apply.
* If the API Gateway is developed by a single team, there can be a development bottleneck. This is another reason why a better approach is to have several fined-grained API Gateways that respond to different client needs. You could also segregate the API Gateway internally into multiple areas or layers that are owned by the different teams working on the internal microservices.