**Microservices**

**The Benefits of Microservices**

* it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services.
* Second, this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract
* Third, the Microservices Architecture pattern enables each microservice to be deployed independently. Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested.
* Finally, the Microservices Architecture pattern enables each service to be scaled independently. You can deploy just the number of instances of each service that satisfy its capacity and availability constraints.

**The Drawbacks of Microservices**

* One drawback is the name itself. The term microservice places excessive emphasis on service size. In fact, there are some developers who advocate for building extremely fine‑grained 10–100 LOC services. While small services are preferable, it’s important to remember that they are a means to an end and not the primary goal
* Another major drawback of microservices is the complexity that arises from the fact that a microservices application is a distributed system. Developers need to choose and implement an inter‑process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure since the destination of a request might be slow or unavailable.
* Another challenge with microservices is the partitioned database architecture. Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices‑based application, however, you need to update multiple databases owned by different services.
* Testing a microservices application is also much more complex. For example, with a modern framework such as Spring Boot it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast, a similar test class for a service would need to launch that service and any services that it depends upon

**Using an API Gateway**

**Direct Client‑to‑Microservice Communication**

* In theory, a client could make requests to each of the microservices directly. Each microservice would have a public endpoint (**https://serviceName.api.company.name**). This URL would map to the microservice’s load balancer, which distributes requests across the available instances. To retrieve the product details, the mobile client would make requests to each of the services listed above.
* Unfortunately, there are challenges and limitations with this option. One problem is the mismatch between the needs of the client and the fine‑grained APIs exposed by each of the microservices. The client in this example has to make seven separate requests.
* Another problem with the client directly calling the microservices is that some might use protocols that are not web‑friendly. One service might use Thrift binary RPC while another service might use the AMQP messaging protocol. Neither protocol is particularly browser‑ or firewall‑friendly and is best used internally. An application should use protocols such as HTTP and WebSocket
* Another drawback with this approach is that it makes it difficult to refactor the microservices. Over time we might want to change how the system is partitioned into services. For example, we might merge two services or split a service into two or more services. If, however, clients communicate directly with the services, then performing this kind of refactoring can be extremely difficult.

**Using an API Gateway**

Usually a much better approach is to use what is known as an [API Gateway](http://microservices.io/patterns/apigateway.html). An API Gateway is a server that is the single entry point into the system. It is similar to the [Facade](https://en.wikipedia.org/wiki/Facade_pattern) pattern from object‑oriented design. The API Gateway encapsulates the internal system architecture and provides an API that is tailored to each client. It might have other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management, and static response handling.

The API Gateway is responsible for request routing, composition, and protocol translation. All requests from clients first go through the API Gateway.

**Benefits and Drawbacks of an API Gateway**

* A major benefit of using an API Gateway is that it encapsulates the internal structure of the application. Rather than having to invoke specific services, clients simply talk to the gateway.
* The API Gateway also has some drawbacks. It is yet another highly available component that must be developed, deployed, and managed. There is also a risk that the API Gateway becomes a development bottleneck. Developers must update the API Gateway in order to expose each microservice’s endpoints.

**Implementing an API Gateway**

* **Performance and Scalability**

Only a handful of companies operate at the scale of Netflix and need to handle billions of requests per day. However, for most applications the performance and scalability of the API Gateway is usually very important. It makes sense, therefore, to build the API Gateway on a platform that supports asynchronous, nonblocking I/O. There are a variety of different technologies that can be used to implement a scalable API Gateway. On the JVM you can use one of the NIO‑based frameworks such Netty, Vertx, Spring Reactor, or JBoss Undertow. One popular non‑JVM option is Node.js, which is a platform built on Chrome’s JavaScript engine.

* **Using a Reactive Programming Model**

The API Gateway handles some requests by simply routing them to the appropriate backend service. It handles other requests by invoking multiple backend services and aggregating the results. With some requests, such as a product details request, the requests to backend services are independent of one another. In order to minimize response time, the API Gateway should perform independent requests concurrently. Sometimes, however, there are dependencies between requests. The API Gateway might first need to validate the request by calling an authentication service, before routing the request to a backend service. Similarly, to fetch information about the products in a customer’s wish list, the API Gateway must first retrieve the customer’s profile containing that information, and then retrieve the information for each product.

* **Service Invocation**

A microservices‑based application is a distributed system and must use an inter‑process communication mechanism. There are two styles of inter‑process communication. One option is to use an asynchronous, messaging‑based mechanism. Some implementations use a message broker such as JMS or AMQP. Others, such as Zeromq, are brokerless and the services communicate directly. The other style of inter‑process communication is a synchronous mechanism such as HTTP or Thrift. A system will typically use both asynchronous and synchronous styles. It might even use multiple implementations of each style. Consequently, the API Gateway will need to support a variety of communication mechanisms.

* **Service Discovery**

The API Gateway needs to know the location (IP address and port) of each microservice with which it communicates. In a traditional application, you could probably hardwire the locations, but in a modern, cloud‑based microservices application this is a nontrivial problem. Infrastructure services, such as a message broker, will usually have a static location, which can be specified via OS environment variables. However, determining the location of an application service is not so easy. Application services have dynamically assigned locations. Also, the set of instances of a service changes dynamically because of autoscaling and upgrades. Consequently, the API Gateway, like any other service client in the system, needs to use the system’s service discovery mechanism: either [Server‑Side Discovery](http://microservices.io/patterns/server-side-discovery.html) or [Client‑Side Discovery](http://microservices.io/patterns/client-side-discovery.html). A [later article](https://www.nginx.com/blog/service-discovery-in-a-microservices-architecture/) will describe service discovery in more detail. For now, it is worthwhile to note that if the system uses Client‑Side Discovery then the API Gateway must be able to query the [Service Registry](http://microservices.io/patterns/service-registry.html), which is a database of all microservice instances and their locations.

* **Handling Partial Failures**

Another issue you have to address when implementing an API Gateway is the problem of partial failure. This issue arises in all distributed systems whenever one service calls another service that is either responding slowly or is unavailable. The API Gateway should never block indefinitely waiting for a downstream service. However, how it handles the failure depends on the specific scenario and which service is failing. For example, if the recommendation service is unresponsive in the product details scenario, the API Gateway should return the rest of the product details to the client since they are still useful to the user. The recommendations could either be empty or replaced by, for example, a hardwired top ten list. If, however, the product information service is unresponsive then API Gateway should return an error to the client.

The API Gateway could also return cached data if that was available. For example, since product prices change infrequently, the API Gateway could return cached pricing data if the pricing service is unavailable. The data can be cached by the API Gateway itself or be stored in an external cache such as Redis or Memcached. By returning either default data or cached data, the API Gateway ensures that system failures do not impact the user experience.

Netflix Hystrix is an incredibly useful library for writing code that invokes remote services. Hystrix times out calls that exceed the specified threshold. It implements a circuit breaker pattern, which stops the client from waiting needlessly for an unresponsive service. If the error rate for a service exceeds a specified threshold, Hystrix trips the circuit breaker and all requests will fail immediately for a specified period of time. Hystrix lets you define a fallback action when a request fails, such as reading from a cache or returning a default value. If you are using the JVM you should definitely consider using Hystrix. And, if you are running in a non‑JVM environment, you should use an equivalent library.