Microservices Design:

The Idea behind microservices architecture is simple, to develop and build a large system, you must decompose its function into relatively small, single-purpose and loosely coupled services. An essential principle of the microservice architecture is loose coupling. If you ignore this principle and develop tightly coupled services the result will mostly likely be yet another “microservices failure story”. Your application will be brittle and have all of disadvantages of both the monolithic and microservice architectures.

Loosely Coupling:

Two systems are loosely coupled if changes to the design, implementation, or behavior in one won’t cause changes in another. When it comes to microservices, coupling can happen if a change to one microservice enforces an almost immediate change to all other microservices that collaborate with it directly or indirectly.

In short, loose coupling in microservice architecture means microservices should know little about each other, and any change to one service should not affect the others.

There are two main types of coupling.

Runtime Coupling:

The first type of coupling is runtime coupling. Runtime coupling is the degree to which the availability of one service is impacted by the availability of another service. Let's imagine that the order service handles a create order request by making a PUT request to the customer service to reserve credit. While this seems simple, it's actually an example of tight runtime coupling. The order service cannot respond to the POST request until it receives a response from the customer service. The availability of the create order endpoint is the product of the availability of both services, which is less than the availability of a single service. This is a simple example of a common antipattern in a distributed application. A good way to eliminate tight runtime coupling is to use asynchronous messaging mechanisms such as the saga pattern. The order service could for example respond immediately to the create request. The response would tell the client that the request to create the order had been received, and that it would need to check back later to determine the outcome. The order service would then exchange messages with the customer service to finalize the creation of the order.

Design Time Coupling:

The second type of coupling is design-time coupling, which is the focus of this talk. Design-time coupling is the degree to which one service is forced to change because of a change to another service. Coupling occurs because one service directly or indirectly depends upon concepts that are owned by another service. Let's imagine that the order service consumes the API of the customer service. It either invokes the services operations, or it subscribes to its events. Dependencies are not inherently bad. Quite often, it's absolutely necessary. However, this creates design-time coupling from the order service to the customer service. The reason design-time coupling is a potential problem is because concepts can change. There is a risk, for example, that a change to the customer service will force this API to change in a way that requires the order service to also change. The degree of coupling is a function of the stability of the customer domain, the design of the customer service API, and how much of that API is consumed by the order service. The tighter the coupling, the greater the risk of lock-step changes. As I describe later, lock-step changes require teams to collaborate, which can reduce productivity.

Consequently, loose coupling is essential. It's important to remember that loose coupling is not guaranteed. You must carefully design your services to be loosely coupled. Ideally, we should design services in a way that avoids any design-time coupling. For example, we might consider turning create order into a local operation by putting the customer and order subdomains in the same service. This might not be a good idea however, if it creates a service that is too large for a small team to maintain. In general, while we can try to avoid design-time coupling, it's usually not practical to eliminate it. Instead, the goal is to minimize it.