nordicapis.com

How to Design Loosely Coupled Microservices I Nordic APIs I

J Simpson

8-11 minutes

Independence is vital for microservices at every stage of their design, development, and deployment. Independent scalability is a core tenant of microservices architecture, enabling different microservices to be emphasized or deprecated depending upon their use.

Decoupled microservices allow for a truly agile development process. Independence and self-sustainability are true to the spirit of the microservice architecture and design ethos. For all of these reasons and more, it's to your advantage to keep your microservices as decoupled as possible.

As API designers, developers, and users, we're used to keeping things separate and independent. Breaking something in the API chain can cause all manner of trouble further down the chain if you're not careful. The same is true for microservices if they're too dependent on one another. To help prevent this from happening, we've put together a few tips on designing, developing, and deploying loosely coupled microservices.

What Is Loose Coupling?

To understand *how* to loosely couple microservices, it's best to start with an idea of what loosely coupled microservices *are* first. Put

simply, loosely cooped microservices are when changes caused to one microservice won't impact another. In a microservices architecture, each microservice should be able to stand on its own.

The concepts behind loosely coupled microservices are fairly mainstream at this point. They're integral to modern development configurations like Agile, DevOps, and CI/CD. On the other hand, monolithic architecture is much more common in the traditional waterfall development model of design > prototype > develop > release.

The fact that we're used to thinking and programming along these lines suggests that this is the perfect time to use loosely coupled microservices if you haven't been already.

How To Create Loosely Coupled Microservices

Here are some tips to try and things to think about that will help you to create your own loosely coupled microservices.

Association Coupling

Microservices may be independent, but they still must work together. When you've got to associate one microservice with another, it's to your benefit to refer back to the origin of the coupling and manage the dependencies instead. For one, this prevents downtime should one component cease functioning. Yes, your users might get some errors, but your app won't go offline entirely.

Managing microservices with dependencies also means that different levels of permissions would be able to access different levels of data, as another example of adopting loosely coupled microservices and permissions instead of tightly coupled microservices.

Use Schemas

One of the main reasons for implementing loosely coupled microservices is to prevent disruptions should something go wrong. Using tightly coupled microservices with downstream microservices too dependent on a specific configuration is a recipe for something wrong.

Instead, try controlling the data you consume with a schema for each microservice. This same approach works for consuming external data as well, as an additional benefit.

Alternate Couplings

Most microservices that use REST will consider response time. Big, complex microservice environments can sometimes be slow to respond, leading to *major* outages if you're not careful. This is reason enough to implement loosely coupled microservices — we want to avoid downtime at all costs.

When developing and deploying your microservice-based apps, you should build fail-safes into your program. This is always a good idea, anyway. It's also relatively straightforward to do using microservices.

When designing microservices, it's typical for each microservice to have its own database. It's not uncommon for data to need to be shared across multiple microservices, however.

The trick is how to stop your users from receiving data that could compromise your database, schema, or the language you use to query your database, all of which could become a security risk. Results based on specific responses could also cause something to break if something changes.

Luckily, APIs are the perfect solution for both of these issues. You can put assets into a JSON file to exchange them to remain secure. Exchanging data with APIs is also better for your customers. For one, they only have to consume the data they're going to use. It

also removes the need to host unnecessary assets, which will keep your apps and services as light and fast as possible.

Keep Dependencies To a Minimum

Sharing data and resources dependent on a particular library can end up as a de facto tight coupling. Instead, you need to ensure that every microservice contains its own dependencies. It also means you should keep shared libraries to a minimum whenever possible. Having to share libraries across numerous microservices can end up causing unnecessary bloat.

This will also make your apps and services language agnostic, as another benefit, as each microservice will be truly as independent and self-sufficient as possible. Queries, responses, and consumed assets can all be formatted using schemas when they're returned.

Asynchronous Communication

Synchronous communication inherently causes temporal coupling. When a caller expects a response from who they're calling in a set amount of time, the time-based decisions built into REST come into play. Implementing asynchronous communication solves this issue immediately.

If you're looking for a simple, stopgap solution to put in place, a circuit breaker configuration will prevent cascading failures from shutting your whole system down. When a response takes too long, the circuit breaker configuration is implemented. You can put a monitoring solution in place when the circuit breaker is tripped as an additional benefit. If you're looking for more of a permanent solution, you can set up an asynchronous communication like Kafka.

Independent Testing Environments

Another of the main purposes of employing microservice architecture is to prevent cascading failures when one service or asset is disrupted. This means that each microservice should strive toward independent deployment as much as possible. Each microservice having its own testing environment is vital to realize that goal.

Avoid Downstream Testing

Speaking of testing environments, it's important to make sure each microservice's test environment is set up properly to avoid outages. Since each microservice needs its own testing environment, try and avoid calling downstream services as much as possible.

If you *do* need to use a service that's downstream of *any* of your microservices, it's a good idea to create a mockup build. If at all possible, make a mockup of your testing service and include it in the same container that's running your microservice. This will also prevent connectivity issues from causing issues with your microservice.

Avoid Domain Creep

One model for implementing microservices is to create a microservice for each business subdomain. This isn't exactly realistic or possible, though, either from a business or a programming angle. The problem lies in selecting what data can be shared and with whom. For every microservice to share all of its data indiscriminately would essentially turn your microservice environment back into a monolith. It also risks exposing sensitive data.

To avoid exposing unnecessary data, try managing data from a single microservice instead. For instance, imagine you've got an eCommerce microservice for taking orders. That microservice needs to interact with both billing and shipping.

In this example, once a customer's payment info has been approved it gets sent on to shipping. Without thinking about it, you might set this up as a linear progression. This is a mistake for numerous reasons, the most important being unnecessarily exposing your customer's billing information.

Instead, you might have the billing service send a confirmation token when the payment's been verified. Then the eCommerce microservice would send the order to the shipping service to be processed.

Final Thoughts On Loosely Coupled Microservices

Microservices have been trending for a number of years now. Tech leads continually hear it's the next big thing and how it's necessary for their success and survival. But the more mainstream and widespread microservices are, the greater the potential they won't be employed correctly.

For something to truly be a microservice, it needs to strive to be as self-sustaining as possible. This is also important to realize in order to take advantage of microservices' benefits.

If you're serious about keeping your apps online with as little downtime as possible, you should implement loosely coupled microservices as much as possible. It will also help to keep your apps lightweight, streamlined, and efficient, which will also help keep your customers happy.