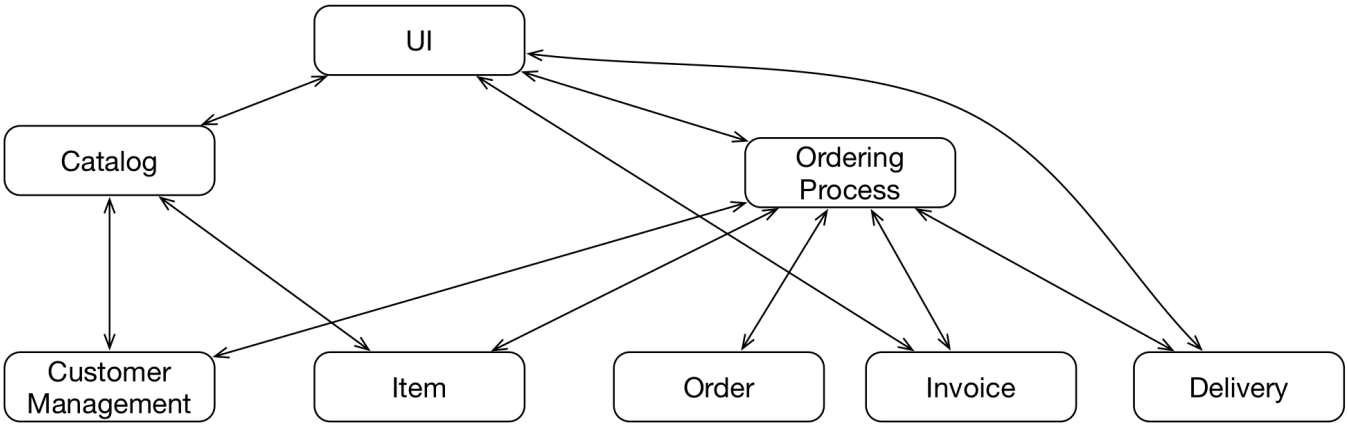
Example

In this lesson, we'll continue to study the details of the example we just saw.

WE'LL COVER THE FOLLOWING

- Consistency
- Disadvantages
- Bounded context
- Tests
- Stubs
- Consumer-driven contract tests
- The pact test framework

Here's the illustration of the example again:



Architecture for a Synchronous System

Consistency

The architecture of this system ensures that **all microservices display the same information about a product or order at all times** because they use synchronous calls to access the respective microservices in which the data is stored.

There is **only one place where each piece of the data is stored**. Every microservice uses that data and therefore shows the latest data.

Disadvantages

However, this architecture also has disadvantages:

- Centralized data storage can lead to problems because the data for displaying the catalog is completely different from that needed for the order process.
- The customer's purchasing behavior is important for the catalog in order to display the correct products.
- For orders, the delivery address or the preferred delivery service are relevant.
- Storing this data in a microservice can make the data model complex.
- *Domain-driven design* states that a domain model is only valid in a particular bounded context, making a centralized model problematic.
- The illustration also reveals another problem; most of the functionalities use many microservices, complicating the system.
- In addition, the failure of a microservice means that many functionalities are not available if no special precautions are taken.
- Performance may also suffer because the microservices have to wait for results from many other microservices.

Bounded context

Such an architecture is often found in synchronous architectures. However, it is not mandatory. Theoretically, it is conceivable that **bounded contexts could communicate synchronously with each other**.

Bounded contexts typically exchange **events**. This is particularly easy with asynchronous communication and **not a great fit for synchronous communication**.

Tests

To enable independent deployment, the tests of each microservice must be **as independent as possible** and integration tests should be kept to a minimum.

For tests in synchronous systems, the communication partners must be available. These can be the microservices used in production.

In this case, setting up an environment is hard because many microservices have to be available, and dependencies arise between the microservices because a new version of a microservice must be made available to all clients in order to enable tests.

Stubs

An alternative are stubs that simulate microservices and simplify the setup of the test environment. The **tests no longer depend on other microservices** because the tests use stubs instead of microservices.

Consumer-driven contract tests

Finally, consumer-driven contract tests can be a solution. With this pattern, the **client writes a test for the server** and can test whether it meets the client's expectations.

This makes it easier to change the server because changes to the interface can be tested without a client. In addition, the **tests no longer depend on the client microservice**.

Consumer-driven contract tests can be written with a **testing framework like JUnit**, for example. The team that implements the called microservice must then execute the tests. If the tests fail, they have made an incompatible change to the microservice.

Either the microservice must be changed to be compatible or the team that wrote the consumer-driven contract test must be informed so that they can change their microservice in such a way that the interface is used differently according to the change. Consumer-driven contract tests therefore formalize the definition of the interface.

As part of the macro architecture, **all teams must agree on a test framework** in which the consumer-driven contract tests are written.

This framework **does not necessarily have to support the programming language in which the microservices are written** when the consumer-driven contract tests use the microservices as black boxes and only access them through the REST interface.

The pact test framework

One option is the framework Pact. It enables writing tests of a REST interface in a programming language. This results in a JSON file containing the REST requests and the expected responses.

With Pact implementations, the JSON file can be interpreted in different programming languages. This increases the technology freedom. An example for Pact with Java can be found at https://github.com/mvitz/pact-example.

QUIZ

1

Suppose you are designing a banking app that requires an incredible degree of consistency. How would you design this app?

COMPLETED 0%

1 of 3

In the next lesson, we'll look at the benefits and challenges of synchronous microservices.