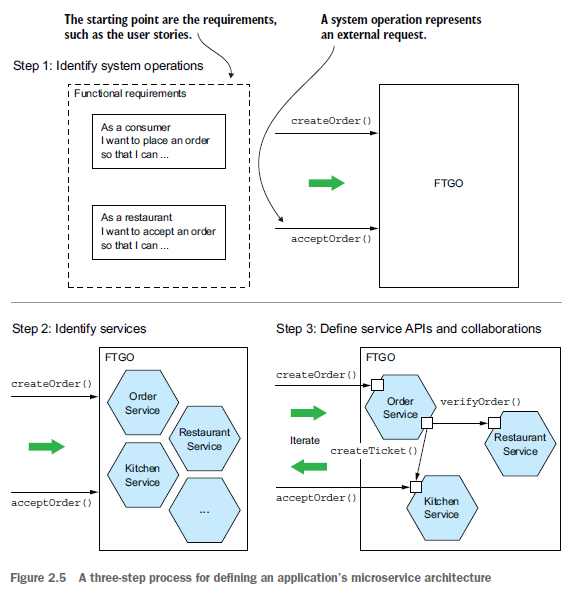
Defining an Application’s Microservices Architecture

This document describes a simple, three-step process, shown in the figure below, for defining an application’s architecture. It’s important to remember, though, that it’s not a process you can follow mechanically. **It’s likely to be iterative and involve a lot of creativity.**



1. An application exists to handle requests, so the first step in defining its architecture is to distill the application’s requirements into the key requests.

Instead of specific IPC(inter-process) technologies like REST or messaging, in this stage we use the more abstract notion of **System Operation**: an abstraction of a request that the application must handle. It’s either a command, which updates data, or a query, which retrieves data. The behavior of each command is defined in terms of an abstract domain model, which is also derived from the requirements. **The system operations become the architectural scenarios that illustrate how the services collaborate.**

1. The second step in the process is to determine the decomposition into services.

There are several strategies to choose from. One strategy, which has its origins in the discipline of business architecture, is to **define services corresponding to business capabilities**. Another strategy is to organize services **around domain-driven design subdomains**. The end result is **services that are organized around business concepts rather than technical concepts.**

1. The third step in defining the application’s architecture is to determine each service’s API.

To do that, **you assign each system operation identified in the first step to a service**. A service might implement an operation entirely by itself. Alternatively, it might need to collaborate with other services. In that case, you determine how the services collaborate, which typically requires services to support additional operations. You’ll also need to decide which of the IPC mechanisms I describe in another document to implement each service’s API.

There are several **obstacles to decomposition:**

* **network latency:** You might discover that a particular decomposition would be impractical due to too many round-trips between services.
* **synchronous communication between services reduces availability**. You might need to use the concept of self-contained services, described in chapter 3.
* **the requirement to maintain data consistency across services**. You’ll typically need to use sagas, discussed in chapter 4.
* **god classes**, which are used throughout an application. Fortunately, you can use concepts from domain-driven design to eliminate god classes.

# Identifying System Operations