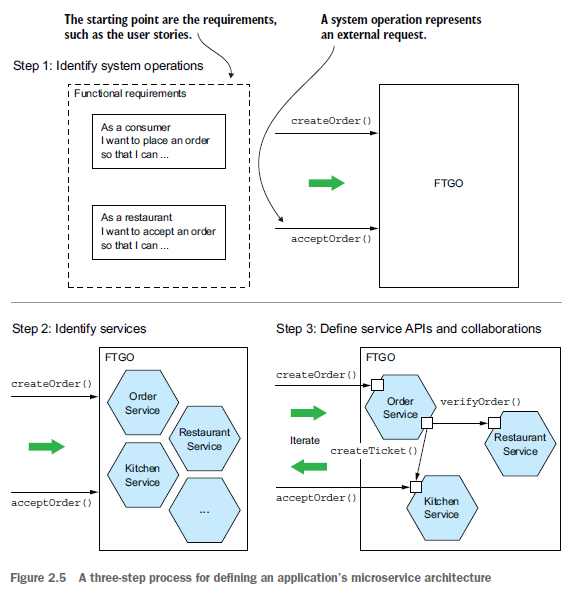
Defining an Application’s Microservices Architecture

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.

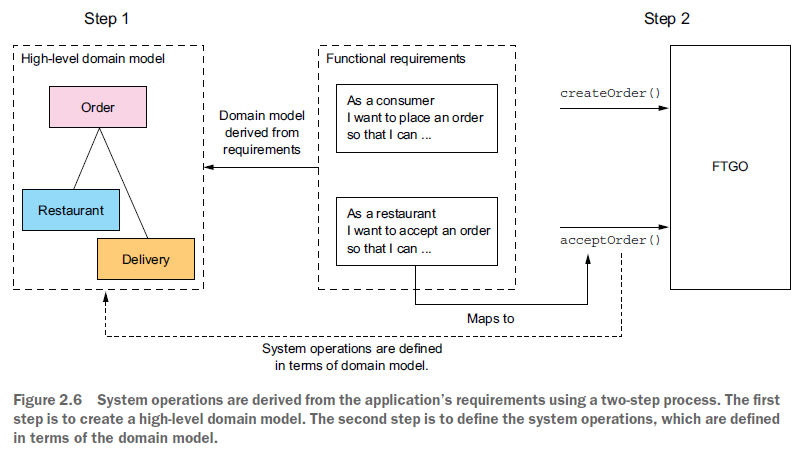
There are several **obstacles to decomposition:**

* **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

**The first step** in defining an **application’s architecture** is to define **the system operations**. **The starting point is the application’s requirements**, including user stories and their associated user scenarios (note that these are different from the architectural scenarios).

The system operations are identified and defined using the two-step process shown in the figure below:



The first step creates the high-level domain model consisting of the **key classes that provide a vocabulary with which to describe the system operations.** The second step identifies the system operations and describes each one’s behavior **in terms of the domain model.**

* The domain model is derived primarily from the nouns of the user stories, and the system operations are derived mostly from the verbs. You could also define the domain model using a technique called Event Storming, which I talk about in chapter 5 of the book.
* The behavior of each system operation is described in terms of its **effect on one or more domain objects and the relationships between them.** A system operation can create, update, or delete domain objects, as well as create or destroy relationships between them.

## Creating a High-Level Domain Model

Note that this domain model is much simpler than what will ultimately be implemented. The application won’t even have a single domain model because, as you’ll soon learn, each service has its own domain model. Despite being a drastic simplification, **a high-level domain model is useful at this stage because it defines the vocabulary for describing the behavior of the system operations.**

domain model is created using standard techniques such as analyzing the nouns in the stories and scenarios and talking to the domain experts. Consider, for example, the **Place Order story**. **We can expand that story into numerous user scenarios including this one:**

Given a consumer

And a restaurant

And a delivery address/time that can be served by that restaurant

And an order total that meets the restaurant's order minimum

When the consumer places an order for the restaurant

Then consumer's credit card is authorized

And an order is created in the PENDING\_ACCEPTANCE state

And the order is associated with the consumer

And the order is associated with the restaurant

The nouns in this user scenario hint at the existence of various classes, including Consumer, Order, Restaurant, and CreditCard.

Similarly, the **Accept Order story** can be expanded into a scenario such as this one:

Given an order that is in the PENDING\_ACCEPTANCE state

and a courier that is available to deliver the order

When a restaurant accepts an order with a promise to prepare by a particular

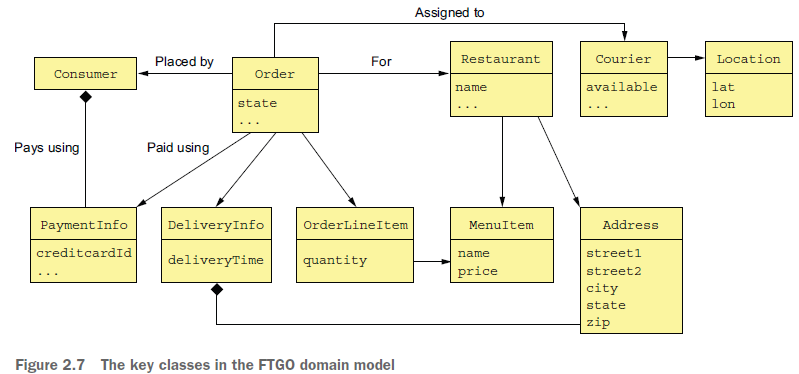
time

Then the state of the order is changed to ACCEPTED

And the order's promiseByTime is updated to the promised time

And the courier is assigned to deliver the order

This scenario suggests the existence of Courier and Delivery classes. The end result after a few iterations of analysis will be a domain model that consists, unsurprisingly, of those classes and others, such as MenuItem and Address. The class diagram below shows the key classes:



The responsibilities of each class are as follows:

 Consumer—A consumer who places orders.

 Order—An order placed by a consumer. It describes the order and tracks its status.

 OrderLineItem—A line item of an Order.

 DeliveryInfo—The time and place to deliver an order.

 Restaurant—A restaurant that prepares orders for delivery to consumers.

 MenuItem—An item on the restaurant’s menu.

 Courier—A courier who deliver orders to consumers. It tracks the availability of

the courier and their current location.

 Address—The address of a Consumer or a Restaurant.

 Location—The latitude and longitude of a Courier.

A class diagram such as the one in figure 2.7 **illustrates one aspect of an application’s architecture.** But it isn’t much more than a pretty **picture without the scenarios to animate it**. The next step is to define **the system operations, which correspond to architectural scenarios.**

## Defining System Operations

Once you’ve defined a high-level domain model, the next step is to identify the requests that the application must handle.

At this point, as different clients might use different communication protocols, we use a more abstract notion of System Operations as described before. Ultimately, these system operations will correspond to REST, RPC, or messaging endpoints, but for now thinking of them abstractly is useful.

Let’s first identify some commands.

A good starting point for identifying system commands is to **analyze the verbs** in the

user stories and scenarios. Consider, for example, the Place Order story. It clearly suggests

that the system must provide a Create Order operation. Many other stories individually

map directly to system commands:

|  |  |  |  |
| --- | --- | --- | --- |
| Actor | Story | Command | Description |
| Consumer | Create Order | createOrder() | Creates an order |
| Restaurant | Accept Order | acceptOrder() | Indicates that the restaurant has accepted the order and is committed to preparing it by the indicated time |
| Restaurant | Order Ready for pickup | noteOrderReadyForPickup() | Indicates that the order is ready for pickup |
| Courier | Update Location | noteUpdatedLocation() | Updates the current location of the  courier |
| Courier | Delivery Picked up | noteDeliveryPickedUp() | Indicates that the courier has picked up the order |
| Courier | Delivery Delivered | noteDeliveryDelivered() | Indicates that the courier has delivered the order |

A command has a specification that defines its parameters, return value, and behavior in terms of the domain model classes. The behavior specification consists of preconditions that must be true when the operation is invoked and mirror the ***given*** part of the user scenario, and post-conditions that are true after the operation is invoked that show ***then*** in the user scenarios. When a system operation is invoked, it **will verify the preconditions** and perform the actions required **to make the post-conditions true**.

Here, for example, is the specification of the createOrder()and acceptOrder() system operations:

**Operation**: createOrder (consumer id, payment method, delivery address, delivery time, restaurant id, order line items)

**Returns**: orderId, …

**Preconditions**:

 The consumer exists and can place orders.

 The line items correspond to the restaurant’s menu items.

 The delivery address and time can be serviced by the restaurant.

**Post-conditions:**

 The consumer’s credit card was authorized for the order total.

 An order was created in the PENDING\_ACCEPTANCE state.

**Operation:** acceptOrder(restaurantId, orderId, readyByTime)

**Returns:** —

**Preconditions:**

 The order.status is PENDING\_ACCEPTANCE.

 A courier is available to deliver the order.

**Post-conditions:**

 The order.status was changed to ACCEPTED.

 The order.readyByTime was changed to the readyByTime.

 The courier was assigned to deliver the order.

Besides implementing commands, an application must also implement queries. The queries provide the UI with the information a user needs to make decisions. At this stage, we don’t have a particular UI design for FTGO application in mind, but consider, for example, the flow when a consumer places an order:

1 User enters delivery address and time.

2 System displays available restaurants.

3 User selects restaurant.

4 System displays menu.

5 User selects item and checks out.

6 System creates order.

This user scenario suggests the following queries:

 findAvailableRestaurants(deliveryAddress, deliveryTime)—Retrieves the restaurants that can deliver to the specified delivery address at the specified time

 findRestaurantMenu(id)—Retrieves information about a restaurant including the menu items

Of the two queries, findAvailableRestaurants() is probably the most architecturally significant:

* It’s a complex query involving geosearch. The geosearch component of the query consists of finding all points—restaurants—that are near a location—the delivery address.
* It also filters out those restaurants that are closed when the order needs to be prepared and picked up.
* **Moreover, performance is critical, because this query is executed whenever a consumer wants to place an order.**

The high-level domain model and the system operations capture what the application does. They help drive the definition of the application’s architecture. The behavior of each system operation is described in terms of the domain model. Each important system operation represents an architecturally significant scenario that’s part of the description of the architecture.

Once the system operations have been defined, the next step is to identify the application’s services. As mentioned earlier, **there isn’t a mechanical process to follow**. There are, however, various **decomposition strategies** that you can use. Each one attacks the problem from a different perspective and uses its own terminology. **But with all strategies, the end result is the same: an architecture consisting of services that are primarily organized around business rather than technical concepts.** Let’s look at the first strategy, which defines services corresponding to business capabilities.

# Identifying Services

## Defining Services by Applying the Decompose by business capability pattern

One strategy for creating a microservice architecture is to decompose by business capability. A concept from business architecture modeling, **a *business capability* is something that a business does in order to generate value**. The set of capabilities for a given business depends on the kind of business. For example, the capabilities of an insurance company typically include Underwriting, Claims management, Billing, Compliance, and so on. **The capabilities of an online store include Order management, Inventory management, Shipping, and so on.**

### Business Capabilities Define What an Application Does

An organization’s business capabilities capture *what* an organization’s business is. **They’re generally stable, as opposed to *how* an organization conducts its business**, which changes over time, sometimes dramatically. That’s especially true today, with the rapidly growing use of technology to automate many business processes.

For example, it wasn’t that long ago that you deposited checks at your bank by handing them to a teller. It then became possible to deposit checks using an ATM. Today you can conveniently deposit most checks using your smartphone. **As you can see, the Deposit check business capability has remained stable, but the manner in which it’s done has drastically changed.**

### Identifying Business Capabilities

An organization’s business capabilities are identified by analyzing the organization’s purpose, structure, and business processes. **Each business capability can be thought of as a service, except it’s business-oriented rather than technical.**

**A business capability is often focused on a particular business object**. For example, **the Claim business object** is the focus of the **Claim management capability**. A capability can often be decomposed **into sub-capabilities**. For example, **the Claim management capability has several sub-capabilities,** including Claim information management, Claim review, and Claim payment management.

the business capabilities for FTGO include the following:

 Supplier management

– *Courier management*—Managing courier information

– *Restaurant information management*—Managing restaurant menus and other information, including location and open hours

 Consumer management—Managing information about consumers

 Order taking and fulfillment

– *Order management*—Enabling consumers to create and manage orders

– *Restaurant order management*—Managing the preparation of orders at a restaurant

– Logistics

– *Courier availability management*—Managing the real-time availability of couriers to delivery orders

– *Delivery management*—Delivering orders to consumers

 Accounting

– *Consumer accounting*—Managing billing of consumers

– *Restaurant accounting*—Managing payments to restaurants

– *Courier accounting*—Managing payments to couriers

 …

The top-level capabilities include **Supplier management**, **Consumer management**, **Order taking and fulfillment**, **and Accounting**. There will likely be many other top-level capabilities, including marketing-related capabilities. Most top-level capabilities are decomposed into sub-capabilities. For example, Order taking and fulfillment is decomposed into five sub-capabilities.

On interesting aspect of this capability hierarchy is that there are three restaurant-related capabilities: Restaurant information management, Restaurant order management, and Restaurant accounting. That’s because they represent three very different aspects of restaurant operations.

### From Business capabilities to Services

Once you’ve identified the business capabilities, you then define a service for **each capability** **or** **group of related capabilities.**

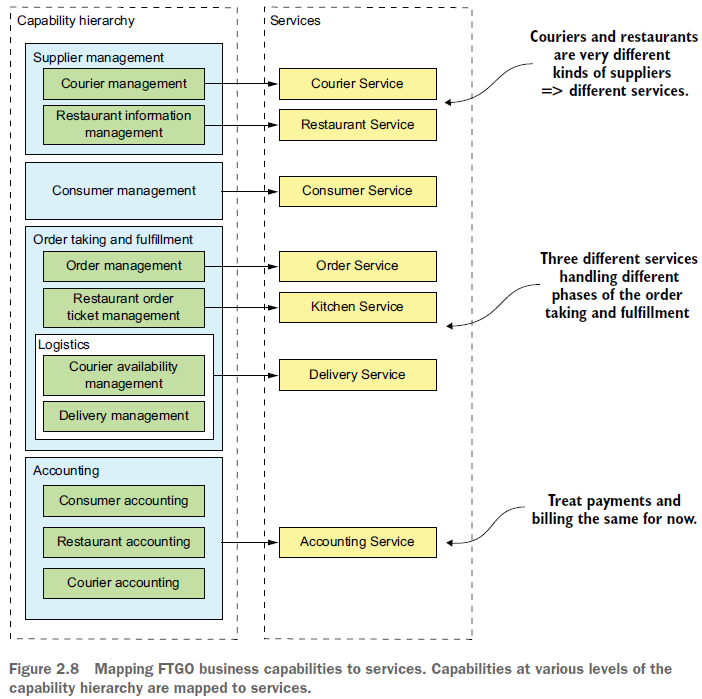
Figure 2.8 shows the mapping from capabilities to services for the FTGO application. **Some top-level capabilities**, such as the Accounting capability, **are mapped to services. In other cases, sub-capabilities are mapped to services.**

The decision of which level of the capability hierarchy to map to services is somewhat subjective. My justification for this particular mapping is as follows:

 I mapped the sub-capabilities of Supplier management to two services, because Restaurants and Couriers are very different types of suppliers.

 I mapped the Order taking and fulfillment capability to three services that are each responsible for different phases of the process. I combined the Courier availability management and Delivery management capabilities and mapped them to a single service because they’re deeply intertwined.

 I mapped the Accounting capability to its own service, because the different types of accounting seem similar.



Later on, it may make sense to separate payments (of Restaurants and Couriers) and billing (of Consumers)**. A key benefit of organizing services around capabilities is that because they’re stable, the resulting architecture will also be relatively stable. The individual components of the architecture may evolve as the *how* aspect of the business changes, but the architecture remains unchanged.**

Having said that, it’s important to remember that the services shown in figure 2.8 are merely the first attempt at defining the architecture. They may evolve over time as we learn more about the application domain. In particular, an important step in the architecture definition process is investigating how the services collaborate in each of the key architectural services.

You might, for example, discover that a particular decomposition is inefficient due to excessive inter-process communication and that you must combine services. Conversely, a service might grow in complexity to the point where it becomes worthwhile to split it into multiple services. What’s more, in section 2.2.5, I describe several obstacles to decomposition that might cause you to revisit your decision.

Let’s take a look at another way to decompose an application that is based on

domain-driven design.

## Defining Services by Applying the Decompose by Sub-Domain Pattern