### **Defining an application’s microservice architecture**

How should we define a microservice architecture? As with any software development effort, the starting points are the written requirements, hopefully domain experts, and perhaps an existing application. Like much of software development, defining an architecture is more art than science.

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.

##### A three-step process for defining an application’s microservice architecture:

##### Diagram Description automatically generated

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. But instead of describing the requests in terms of specific IPC technologies such as REST or messaging, I use the more abstract notion of system operation. A system operation is 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.

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.

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.

There are several obstacles to decomposition. The first is network latency. You might discover that a particular decomposition would be impractical due to too many round-trips between services.

Another obstacle to decomposition is that synchronous communication between services reduces availability. You might need to use the concept of self-contained services.

The third obstacle is the requirement to maintain data consistency across services. You’ll typically need to use sagas

The fourth and final obstacle to decomposition is so-called god classes, which are used throughout an application. Fortunately, you can use concepts from domain-driven design to eliminate god classes.

**Identifying 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, show below:

Diagram

Description automatically generated

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 (Will be discussed in Chapter 5).

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

The first step in the process of defining the system operations is to sketch a high-level domain model for the application. 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.

A 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.

##### Diagram Description automatically generated

The responsibilities of each class are as follows:

Text

Description automatically generated

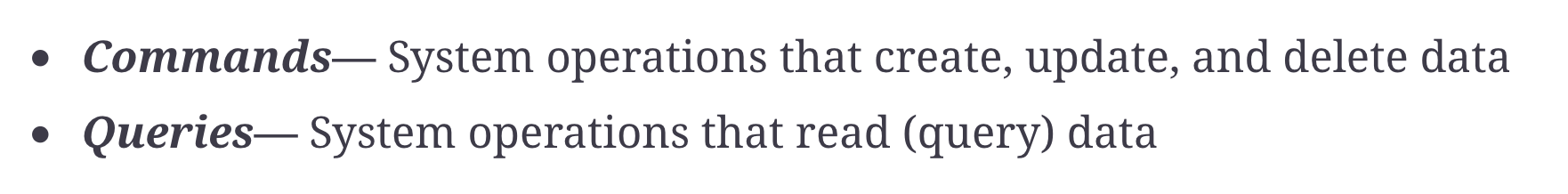
The next step is to define the system operations, which correspond to architectural scenarios.

##### 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.

The UI will make requests to the backend business logic to retrieve and update data. FTGO is primarily a web application, which means that most requests are HTTP-based, but it’s possible that some clients might use messaging. Instead of committing to a specific protocol, therefore, it makes sense to use the more abstract notion of a system operation to represent requests.

There are two types of system operations:



Ultimately, these system operations will correspond to REST, RPC, or messaging endpoints, but for now thinking of them abstractly is useful.

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 post-conditions that are true after the operation is invoked.

Most of the architecturally relevant system operations are commands. Sometimes, though, queries, which retrieve data, are also important.

Besides implementing commands, an application must also implement queries.

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.

**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.

**Pattern: Decompose by business capability**

Define services corresponding to business capabilities. See <http://microservices.io/patterns/decomposition/decompose-by-business-capability.html>.

##### Business capabilities define what an organization 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.

##### 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. Its specification consists of various components, including inputs, outputs, and service-level agreements.A business capability is often focused on a particular business object.

##### 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.

##### Services 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 interprocess 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.

##### Defining services by applying the Decompose by sub-domain pattern

DDD(Domain-driven design) is an approach for building complex software applications that is centered on the development of an object-oriented domain model. A domain mode captures knowledge about a domain in a form that can be used to solve problems within that domain. It defines the vocabulary used by the team, what DDD calls the Ubiquitous Language. The domain model is closely mirrored in the design and implementation of the application. DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts.

**Pattern: Decompose by subdomain**

Define services corresponding to DDD subdomains. See <http://microservices.io/patterns/decomposition/decompose-by-subdomain.html>.

2.2. Defining an application’s microservice architecture

How should we define a microservice architecture? As with any software development effort, the starting points are the written requirements, hopefully domain experts, and perhaps an existing application. Like much of software development, defining an architecture is more art than science. This section describes a simple, three-step process, shown in figure 2.5, 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.

Figure 2.5. A three-step process for defining an application’s microservice architecture

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. But instead of describing the requests in terms of specific IPC technologies such as REST or messaging, I use the more abstract notion of system operation. A system operation is 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.

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.

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 chapter 3 to implement each service’s API.

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

This section first describes how to identity an application’s operations. After that, we’ll look at strategies and guidelines for decomposing an application into services, and at obstacles to decomposition and how to address them. Finally, I’ll describe how to define each service’s API.

2.2.1. Identifying the 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 figure 2.6. This process is inspired by the object-oriented design process covered in Craig Larman’s book Applying UML and Patterns (Prentice Hall, 2004) (see www.craiglarman.com/wiki/index.php?title=Book\_Applying\_UML\_and\_Patterns for details). 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.

Figure 2.6. System operations are derived from the application’s requirements using a two-step process. The first step is to create a high-level domain model. The second step is to define the system operations, which are defined 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. 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.

Let’s look at how to define a high-level domain model. After that I’ll define the system operations in terms of the domain model.

Creating a high-level domain model

The first step in the process of defining the system operations is to sketch a high-level domain model for the application. 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.

A 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. Figure 2.7 is a class diagram that shows the key classes.

Figure 2.7. The key classes in the FTGO domain model

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. The details of the UI are beyond the scope of this book, but you can imagine that in each user scenario, the UI will make requests to the backend business logic to retrieve and update data. FTGO is primarily a web application, which means that most requests are HTTP-based, but it’s possible that some clients might use messaging. Instead of committing to a specific protocol, therefore, it makes sense to use the more abstract notion of a system operation to represent requests.

There are two types of system operations:

Commands— System operations that create, update, and delete data

Queries— System operations that read (query) data

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. Table 2.1 lists some of the key system commands.

Table 2.1. Key system commands for the FTGO application

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 post-conditions that are true after the operation is invoked. Here, for example, is the specification of the createOrder() system operation:

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.

The preconditions mirror the givens in the Place Order user scenario described earlier. The post-conditions mirror the thens from the scenario. When a system operation is invoked it will verify the preconditions and perform the actions required to make the post-conditions true.

Here’s the specification of the acceptOrder() system operation:

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.

Its pre- and post-conditions mirror the user scenario from earlier.

Most of the architecturally relevant system operations are commands. Sometimes, though, queries, which retrieve data, are also important.

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:

User enters delivery address and time.

System displays available restaurants.

User selects restaurant.

System displays menu.

User selects item and checks out.

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.

2.2.2. 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.

Pattern: Decompose by business capability

Define services corresponding to business capabilities. See http://microservices.io/patterns/decomposition/decompose-by-business-capability.html.

Business capabilities define what an organization 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. Its specification consists of various components, including inputs, outputs, and service-level agreements. For example, the input to an Insurance underwriting capability is the consumer’s application, and the outputs include approval and price.

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.

It is not difficult to imagine that 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.

One 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.

Next we’ll look at how to use business capabilities to define services.

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.

Figure 2.8. Mapping FTGO business capabilities to services. Capabilities at various levels of the capability hierarchy 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 interprocess 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.

2.2.3. Defining services by applying the Decompose by sub-domain pattern

DDD, as described in the excellent book Domain-driven design by Eric Evans (Addison-Wesley Professional, 2003), is an approach for building complex software applications that is centered on the development of an object-oriented domain model. A domain mode captures knowledge about a domain in a form that can be used to solve problems within that domain. It defines the vocabulary used by the team, what DDD calls the Ubiquitous Language. The domain model is closely mirrored in the design and implementation of the application. DDD has two concepts that are incredibly useful when applying the microservice architecture: subdomains and bounded contexts.

Pattern: Decompose by subdomain

Define services corresponding to DDD subdomains. See [http://microservices.io/patterns/decomposition/decompose-by-subdomain.html.](DDD%20defines%20a%20separate%20domain%20model%20for%20each%20subdomain.%20A%20subdomain%20is%20a%20part%20of%20the%20domain,%20DDD’s%20term%20for%20the%20application’s%20problem%20space.%20Subdomains%20are%20identified%20using%20the%20same%20approach%20as%20identifying%20business%20capabilities:%20analyze%20the%20business%20and%20identify%20the%20different%20areas%20of%20expertise.%20The%20end%20result%20is%20very%20likely%20to%20be%20subdomains%20that%20are%20similar%20to%20the%20business%20capabilities.)

DDD is quite different than the traditional approach to enterprise modeling, which creates a single model for the entire enterprise. In such a model there would be, for example, a single definition of each business entity, such as customer, order, and so on. The problem with this kind of modeling is that getting different parts of an organization to agree on a single model is a monumental task. Also, it means that from the perspective of a given part of the organization, the model is overly complex for their needs. Moreover, the domain model can be confusing because different parts of the organization might use either the same term for different concepts or different terms for the same concept. DDD avoids these problems by defining multiple domain models, each with an explicit scope.

DDD defines a separate domain model for each subdomain. A subdomain is a part of the domain, DDD’s term for the application’s problem space. Subdomains are identified using the same approach as identifying business capabilities: analyze the business and identify the different areas of expertise. The end result is very likely to be subdomains that are similar to the business capabilities.

DDD calls the scope of a domain model a bounded context. A bounded context includes the code artifacts that implement the model. When using the microservice architecture, each bounded context is a service or possibly a set of services. We can create a microservice architecture by applying DDD and defining a service for each subdomain.

Diagram

Description automatically generated

Decompose by subdomain and Decompose by business capability are the two main patterns for defining an application’s microservice architecture. There are, however, some useful guidelines for decomposition that have their roots in object-oriented design. Let’s take a look at them.

#### **Decomposition guidelines**

##### Single Responsibility Principle

One of the main goals of software architecture and design is determining the responsibilities of each software element. Each responsibility that a class has is a potential reason for that class to change. If a class has multiple responsibilities that change independently, the class won’t be stable. By following the SRP, you define classes that each have a single responsibility and hence a single reason for change. We can apply SRP when defining a microservice architecture and create small, cohesive services that each have a single responsibility. This will reduce the size of the services and increase their stability.

##### Common Closure Principle

##### Text Description automatically generated

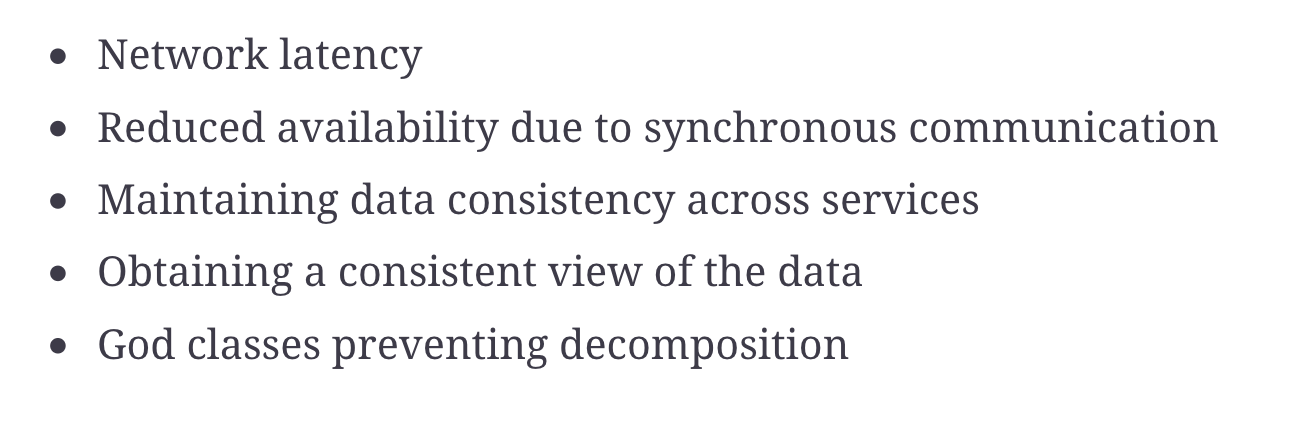
The idea is that if two classes change in lockstep because of the same underlying reason, then they belong in the same package. Perhaps, for example, those classes implement a different aspect of a particular business rule. The goal is that when that business rule changes, developers only need to change code in a small number of packages (ideally only one). Adhering to the CCP significantly improves the maintainability of an application.

We can apply CCP when creating a microservice architecture and package components that change for the same reason into the same service. Doing this will minimize the number of services that need to be changed and deployed when some requirement changes. Ideally, a change will only affect a single team and a single service. CCP is the antidote to the distributed monolith anti-pattern.

Decomposition by business capability and by subdomain along with SRP and CCP are good techniques for decomposing an application into services. In order to apply them and successfully develop a microservice architecture, you must solve some transaction management and interprocess communication issues.

**Obstacles to decomposing an application into services**

On the surface, the strategy of creating a microservice architecture by defining services corresponding to business capabilities or subdomains looks straightforward. You may, however, encounter several obstacles:



##### Network latency

*Network latency* is an ever-present concern in a distributed system. You might discover that a particular decomposition into services results in a large number of round-trips between two services. Sometimes, you can reduce the latency to an acceptable amount by implementing a batch API for fetching multiple objects in a single round trip. But in other situations, the solution is to combine services, replacing expensive IPC with language-level method or function calls.

##### Synchronous interprocess communication reduces availability

##### Another problem is how to implement interservice communication in a way that doesn’t reduce availability. For example, the most straightforward way to implement the createOrder() operation is for the Order Service to synchronously invoke the other services using REST. The drawback of using a protocol like REST is that it reduces the availability of the Order Service. It won’t be able to create an order if any of those other services are unavailable. Sometimes this is a worthwhile trade-off, using asynchronous messaging, which eliminates tight coupling and improves availability, is often a better choice.

##### Maintaining data consistency across services

Another challenge is maintaining data consistency across services. Some system operations need to update data in multiple services. For example, when a restaurant accepts an order, updates must occur in both the Kitchen Service and the Delivery Service. The Kitchen Service changes the status of the Ticket. The Delivery Service schedules delivery of the order. Both of these updates must be done atomically.

The traditional solution is to use a two-phase, commit-based, distributed transaction management mechanism. This is not a good choice for modern applications, and you must use a very different approach to transaction management, a saga. A *saga* is a sequence of local transactions that are coordinated using messaging. Sagas are more complex than traditional ACID transactions but they work well in many situations. One limitation of sagas is that they are eventually consistent. If you need to update some data atomically, then it must reside within a single service, which can be an obstacle to decomposition.

##### Obtaining a consistent view of the data

##### Another obstacle to decomposition is the inability to obtain a truly consistent view of data across multiple databases. In a monolithic application, the properties of ACID transactions guarantee that a query will return a consistent view of the database. In contrast, in a microservice architecture, even though each service’s database is consistent, you can’t obtain a globally consistent view of the data. If you need a consistent view of some data, then it must reside in a single service, which can prevent decomposition. Fortunately, in practice this is rarely a problem.

##### God classes prevent decomposition

##### Another obstacle to decomposition is the existence of so-called god classes. God classes are the bloated classes that are used throughout an application (http://wiki.c2.com/?GodClass). A god class typically implements business logic for many different aspects of the application. It normally has a large number of fields mapped to a database table with many columns. Most applications have at least one of these classes, each representing a concept that’s central to the domain: accounts in banking, orders in e-commerce, policies in insurance, and so on. Because a god class bundles together state and behavior for many different aspects of an application, it’s an insurmountable obstacle to splitting any business logic that uses it into services.

The Order class is a great example of a god class in the FTGO application. That’s not surprising—after all, the purpose of FTGO is to deliver food orders to customers. Most parts of the system involve orders. If the FTGO application had a single domain model, the Order class would be a very large class. It would have state and behavior corresponding to many different parts of the application.

##### The Order god class is bloated with numerous responsibilities.

##### Diagram Description automatically generated

As you can see, the Order class has fields and methods corresponding to order processing, restaurant order management, delivery, and payments. This class also has a complex state model, due to the fact that one model has to describe state transitions from disparate parts of the application. In its current form, this class makes it extremely difficult to split code into services.

One solution is to package the Order class into a library and create a central Order database.

All services that process orders use this library and access the access database. The trouble with this approach is that it violates one of the key principles of the microservice architecture and results in undesirable, tight coupling. For example, any change to the Order schema requires the teams to update their code in lockstep.

Another solution is to encapsulate the Order database in an Order Service, which is invoked by the other services to retrieve and update orders. The problem with that design is that the Order Service would be a data service with an anemic domain model containing little or no business logic. Neither of these options is appealing, but fortunately, DDD provides a solution.

A much better approach is to apply DDD and treat each service as a separate subdomain with its own domain model. This means that each of the services in the FTGO application that has anything to do with orders has its own domain model with its version of the Order class. A great example of the benefit of multiple domain models is the Delivery Service.

Its view of an Order, shown is extremely simple: pickup address, pickup time, delivery address, and delivery time. Moreover, rather than call it an Order, the Delivery Service uses the more appropriate name of Delivery.

Diagram

Description automatically generated

The Delivery Service isn’t interested in any of the other attributes of an order.

**Defining service APIs**

So far, we have a list of system operations and a list of a potential services. The next step is to define each service’s API: its operations and events. A service API operation exists for one of two reasons: some operations correspond to system operations. They are invoked by external clients and perhaps by other services. The other operations exist to support collaboration between services. These operations are only invoked by other services.

The starting point for defining the service APIs is to map each system operation to a service. After that, we decide whether a service needs to collaborate with others to implement a system operation. If collaboration is required, we then determine what APIs those other services must provide in order to support the collaboration. Let’s begin by looking at how to assign system operations to services.

##### Assigning system operations to services

The first step is to decide which service is the initial entry point for a request. Many system operations neatly map to a service, but sometimes the mapping is less obvious.

Determining the APIs required to support collaboration between services. Some system operations are handled entirely by a single service.