Apply Domain-Driven Design to microservices architecture

Domain-Driven Design provides concepts to help you get started using microservices for applications.

You understand the benefits of using microservices for new applications and refactoring applications to microservices, but do you know where to start? How do you design the microservices that your application needs? What are the elements of a good microservices design? The answers lie in some of the basic premises of object-oriented programming.

The book *Domain-Driven Design: Tackling Complexity in the Heart of Software* (Evans 2004) captures a meta-process for designing software that object-oriented software development teams have used for years. The book isn't about specific design notations or even specific classes of objects or patterns. Instead, it covers the general categories of objects that good object-oriented designers identify and work with. Those categories become critical elements of a good microservices design.

Entities and their extension Aggregates

Start your microservices design with the idea of an Entity. An *Entity* is an object that is distinguished by its identity. Entities have unique identifiers. Entities also have attributes that might change, but the identifier for the person stays the same. An example of an entity is a person. A person has an unchanging identifier, such as a Social Security Number in the United States. That same person has a given name, a surname, an address, and a phone number. Any of the attributes can change, but the person doesn't change.

Evans notes that entity objects must have a well-defined lifecycle and a good definition of what the identity relationship is—what it means to be the same thing as another thing.

From Entity-Relationship modeling, you know that sometimes Entities might be well-defined and have a specific well-known identifier, but might not live independently. Evans calls the combination of Entities an *Aggregate*.

You can find a simple example of the Entity/Aggregate relationship in a retail store. If you go to the soda aisle in a grocery store, you can buy a 12-pack carton of soda. Each can in the carton has a bar code to identify it individually, but you can't buy a single can from the carton. The cans are entities that are referred to as Dependent Entities. The root entity is the carton. The carton is the root of the Aggregate because the root defines the Dependent Entities' lifecycle.

Value Objects

The opposite of an Entity is a Value Object. Value Objects have no conceptual identity. You can't tell one Value Object from any other of its type. If you treat all objects in a system as Entities, the process to assess and manage the identity of each object can become overwhelming and hurt performance. Instead, care only about the attributes of a Value Object and that each Value Object can be treated as unchanged from the creation of the object until it is destroyed.

Consider an example of a Value Object. When you go to the bank, you usually want to access your bank account. Your bank account is an entity. If you query your current account balance, the result is a Value Object. The next time that you query your account balance, a different Value Object is returned with your new current balance.

Repositories

Another important concept is the idea of a Repository. Here's the issue: Aggregates and non-aggregate Entities must be discovered through a search based on attributes. When you provide a search mechanism, you want to hide the details of the technical database infrastructure that

implements search. The Repository is the thing that "maintains the illusion" that all objects are in-memory and instantly searchable while it hides the messiness of the database implementation behind it.

Services

In most domains, some operations don't conceptually belong to any specific object. Previous design methods tried to force those operations into an entity-based model, often with adverse consequences, especially for operations that operated on a group of related Entities. Instead, you model those objects as stand-alone interfaces called *Services*. The rules for a good service are as follows:

- It is stateless
- The interface is defined in terms of other elements of your domain model (Entities and Value Objects)
- It refers to a domain concept that doesn't naturally correspond to any particular Entity or Value Object

These operations are part of the business domain—they're not technical issues of implementation. Things like "Login," "Authentication," and "Logging" aren't appropriate Services of this type. However, a concept like "Funds Transfer" in banking or "Adjudication" in insurance might be.

Put it all together

When you follow the Domain-Driven Design process, you end up with these object types:

- A list of Entities, some of which are Aggregates, including identified root Aggregates and Entities
- Defined Repositories
- A list of Value Objects that are associated with one or more Entities
- A list of Services that correspond to functions that aren't part of any particular Entity

These object types are what you need to create your first-pass RESTful service design for microservices. Not all of these objects become part of your microservices design—some might be hidden in your service implementation—but you can start to understand the objects that you're dealing with.

Which comes first: the API or the objects?

An ongoing argument exists about which comes first: the design of your API or the design of the Objects that implement your API. Many early distributed-computing proponents advocated designing the Objects first and then making your API the same as your Object API. This approach led to problems in the granularity of the API, which often resulted in APIs that represented technical interfaces instead of business interfaces.

When you start with the API, you can focus on solving the business problem and avoid getting lost in the technical details of a particular implementation. Pact testing is a slightly different version of API-first. Pact is a contract unit-testing tool that ensures that services can communicate with each other.

When you write pact tests, you unit-test the consumer and the provider separately. To test the consumer of an API, you submit an actual API request to a mock provider and receive a response. To test the provider, a mock consumer issues a request and the provider provides an actual response. Verification is done in the mocked code to ensure that the services work as expected. Testing is done only on the specific functions that the customer uses.

The next consideration when you design an API is whether your APIs represent Entities or Functions. The starting point for mapping is the Resource API pattern (Daigneau 2011). This pattern is a stripped-down definition of the central idea of REST. You assign all procedures and instances of domain data a URI and then use the application protocol of HTTP to define the standard service behaviors by using standard status codes wherever possible. In this model, a request consists of a URI, an HTTP server method, and optionally, and rarely, a Media type. That combination uniquely selects the particular Service that fulfills the request.

The key notion is that each URI represents a Resource, not a procedure. Resource APIs must be the bulk of the APIs that you specify. They're the rule rather than the exception.

However, exceptions exist. The best way to handle non-entity Services is to think through the problem of "noun-ifying" them with the Process API pattern. A Process API is a reification of an action in a domain. If you can name a Service as a noun in Domain-Driven Design, that Service name becomes the URI path. A good rule is that if a paper-and-pencil version of the thing exists, it can become a resource, but if it’s something a person must do, it becomes a process. An example of a process API might be processing a purchase order.

When you start with Domain-Driven Design, you find that the large-grained concepts that are derived through the process are closer to the right level for a good API design:

- Aggregates and Entities become Resource APIs
- Value Objects inform the design of the schema that the Resource API uses

- Services become Process APIs

Define boundaries

You can apply one final Domain-Domain Driven Design concept to microservices design. A Bounded Context explicitly defines the boundaries of your model. This concept is critical in large software projects. A Bounded Context sets the limits around what a specific team works on and helps them to define their own vocabulary within that particular context.

Take the example of a customer in retail. In one context, a customer is a person who buys products from a store. In another context, a customer is a person who a retailer markets its products to. The customer is the same but is in different bounded contexts, each of which act on the customer entity based on their own rules.

When you define a bounded context, you define who uses it, how they use it, where it applies within a larger application context, and what it consists of in terms of things like Swagger documentation and code repositories.

In summary, when you design microservices for an application, use the principles of Domain-Driven Design to guide you along the way. Establish the Bounded Context for your team and list your Entities, Repositories, Value Objects, and Services. Then, use what you learned to define and design your microservices.

References

- Daigneau, Robert. *Service Design Patterns: Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services*. Boston: Addison-Wesley, 2011.
- Evans, Eric. *Domain Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004.

Kyle Brown

IBM, IBM Fellow, CTO Cloud Architecture, IBM Cloud Garage and Solution Engineering

---