Defining Microservices Boundaries

One of the most popular definition we see of microservices these days is that a microservices does one thing and one thing well. This is probably the most dangerous view of a microservice there is. The reason for that is there is no clear definition of what is that one thing; how big or how small. Let’s take the example of a calculator. Does performing arithmetic operations considered a thing? It seems to be many things: addition, subtraction, multiplication, division, etc. What about the operations on complexes numbers? Or binary? If we don’t properly define the boundaries we will end up with microservices that have hidden dependencies and that are highly coupled together. Instead, they should be loosely coupled to each other’s and high functional cohesion. Hence, they should be designed around business capabilities.

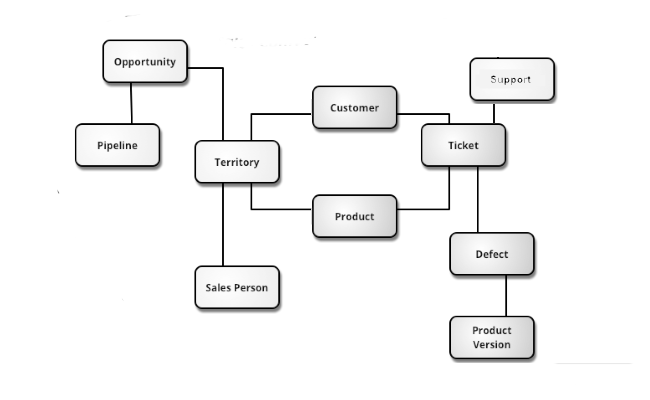
# Domain Analysis

One way to achieve correct microservices boundaries is to perform a domain-driven design analysis on the application to be written. As with many things in software development, this is an iterative process and it involves the following steps:

1. Define the functional requirements
2. Define the bounded contexts
3. Define the domain models
4. Define the microservices

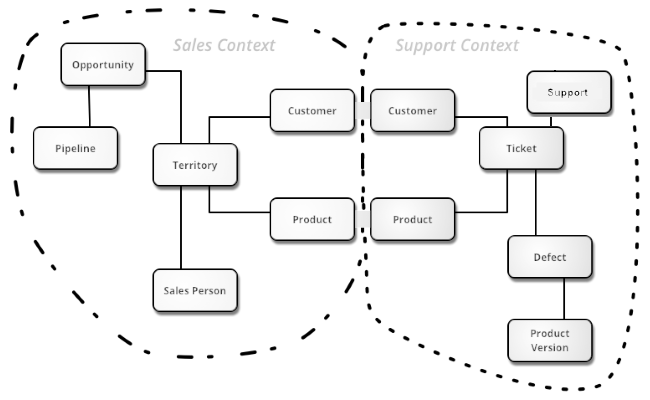
## Defining Functional Requirements

In order to define the functional requirements, we want to map all the business functions and how they are connected to each other. Technologies and implementation details are not important at this stage but you should identify dependencies towards external systems. A map might look like the following sketch taken and adapted from <https://martinfowler.com/bliki/BoundedContext.html> where we have a very basic analysis of selling products and providing support for sold products.



## Defining Bounded Contexts

The next step is to define the bounded contexts. A bounded context is simply the boundary where the domain model makes sense. For example, if within the same application, the functionality of “product” differs then you most likely have multiple contexts. If we look at the sketch below from <https://martinfowler.com/bliki/BoundedContext.html>, both “customer” and “product” can have different properties since they aim to resolve different functional requirements (sale and support). A product during sale might include information on inventory whereas for support the information would be irrelevant. In the case of product replacement, it would be considered a sale at no cost.



## Defining Domain Models

The following steps is to define the domain models in greater details by clearly identifying the properties and behaviors. It is also important to note that this exercise must be applied within a single bounded context. Similarly, it is also possible that, during this exercise, new bounded context emerges. During this exercise, we will look to identify the following:

* Entities: An entity is an object that is uniquely identified, persisted and mutable. Typical examples: persons, accounts, products.
* Value Objects: A value object is without identify and immutable. Typical examples: colors, currencies, dates and times.
* Aggregates: Aggregate define a logical boundary around one or more entities where one of the entities is the root. Typical examples: Car (with parts).
* Domain Services: Domain services encapsulate domain logic involving multiple aggregates without holding any state. Typical examples: scheduler
* Application Services: Application services provide a technical functionality to the application. Typical examples: user authentication, sending an email.
* Domain Events: Domain events are used to perform notification to inform the rest of the application that something of interest occurred.

## Defining Microservices Boundaries

Having a well-designed domain model will make this step much easier. As a general rule of thumb, a microservice should not be bigger than a single bounded context and smaller than an aggregate. Hence, they could be the size of a single bounded context, a domain service or an aggregate. Since it is easier to slice things off than to put things back together, it is recommended to start with the bounded context and divide further only when necessary. The following criteria can be used to validate the design:

* A microservice has a single responsibility
* Communication between microservices is not chatty. If it is the case, they should be part of the same service
* A microservice can be deployed without redeploying other services
* There will be no data inconsistency between microservices. If it is the case, they should be part of the same service

Data consistency is something we will look into greater details in a future post.

# References

<https://docs.microsoft.com/en-us/azure/architecture/microservices/domain-analysis>

<https://docs.microsoft.com/en-us/azure/architecture/microservices/microservice-boundaries>