Table of Contents

Integrating with External Systems

> Wholeness is not achieved by cutting off a portion of one's being, but by integration of the contraries.
>
> — Carl Jung

Thus far, we have used DDD to implement a robust core for our application. However, most bounded contexts usually have both upstream and downstream dependencies which usually change at a pace which is different from these core components. To maintain both agility and reliability and enable loose coupling, it is important to create what DDD calls the anti-corruption layer in order to shield the core from everything that surrounds it. In this chapter, we will look at integrating with a legacy Inventory Management system. We will round off by looking at common patterns when integrating with legacy applications.

Technical Requirements

Continuing our design journey

From our domain analysis in the earlier chapters, we have arrived at four bounded contexts for our application as depicted here:

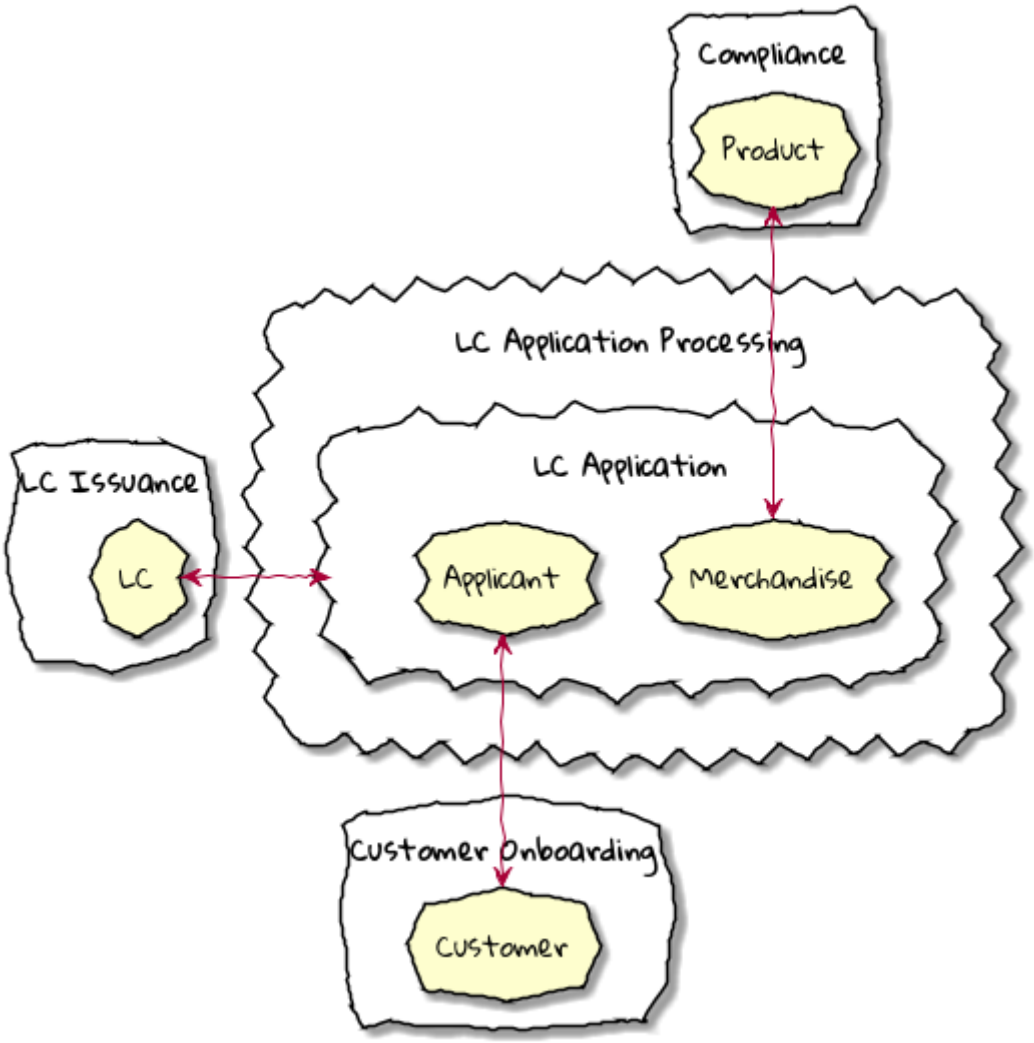

Figure 1. Context map for the LC solution

Thus far, our focus has been on the implementation of the internals of the **LC Application** bounded context. While the LC Application bounded context is independent of the other bounded contexts, it is not completely isolated from them. For example, when processing an LC application, we need to perform merchandise and applicant checks which require interactions with the **Compliance** and **Customer Onboarding** bounded contexts respectively. This means that these bounded contexts have a relationship with each other. These relationships are driven by the nature of collaboration between the teams working on the respective bounded contexts. Let's examine how these team dynamics influence integration mechanisms between bounded contexts in a way that continues to preserve their individual integrity.

Bounded context relationships

We need bounded contexts to as independent as possible. However, this does not mean that bounded contexts are completely isolated from each other. Bounded contexts need to collaborate with others to provide business value. Whenever there is collaboration required between two bounded contexts, the nature of their relationship is not only influenced by their individual goals and priorities, but also by the prevailing organizational realities.In a high performing environment, it is fairly common to have a single team assume ownership of a bounded context. The relationships between the teams owning these bounded contexts, play a significant role in influencing the integration patterns employed to arrive at a solution. At a high level, there are two

categories of relationships:

1. Symmetric
2. Asymmetric

Let's look at these relationship types in more detail.

Symmetric relationship patterns

Two teams can be said to have a symmetric relationship when they have an equal amount of influence in the decision-making process to arrive at a solution. Both teams are in a position to and indeed, do contribute more or less equally towards the outcome. There are three variations of symmetric relationships, each of which we outline in more detail here:

Partnership

In a partnership, both teams integrate in an adhoc manner. There are no fixed responsibilities assigned when needing complete integration work. Each team picks up work as and when needed without the need for any specific ceremony or fanfare. The nature of the integration is usually two-way with both teams exchanging solution artifacts as and when needed. Such relationships require extremely high degrees of collaboration and understanding of the work done by both teams.

For example, a web front-end team working in close collaboration with the APIs team building the BFFs for the front-end. The BFF team creates experience APIs meant to be used exclusively by the front-end. To fulfill any functionality, the front-end team requires capabilities to be exposed by the APIs team. On the other hand, the APIs team is dependent on the front-end team to provide advice on what capabilities to build and the order in which to build them. Both teams freely make use of each other's domain models (for example, the same set of request and response objects that define the API) to implement functionality. Such reuse happens mostly arbitrarily and when API changes happen, the both teams coordinate changes to keep things working.

Shared kernel

Separate ways

Asymmetric relationship patterns

Conformist

Anti-corruption layer

Open host service

Implementation patterns

Data-based

API-based

HTTP-based APIs

Message-based APIs

Shared code artifacts

Enforcing contracts

Legacy Application Migration Patterns