23

Architecting Application User Interfaces

WHAT'S IN THIS CHAPTER?

- An introduction to the UI challenges involved in loosely coupled, distributed, and nondistributed Domain-Driven Design (DDD) systems
- An example of building a UI that pulls in content from multiple bounded contexts that run as a single application
- An example of building a UI that pulls in content from distributed bounded contexts

WROX.COM CODE DOWNLOADS FOR THIS CHAPTER

The wrox.com code downloads for this chapter are found at `www.wrox.com/go/domaindrivendesign` on the Download Code tab. The code is in the Chapter 23 download and individually named according to the names throughout the chapter.

Customers mostly care about the user interface of your application. If it looks compelling and allows them to achieve what they want, such as finding the perfect holiday, they will be happy to spend lots of money. But getting the UI right is more than just about letting designers come up with eye candy. There are significant engineering challenges tied to the performance, scalability, and loose coupling of your behind-the-scenes bounded contexts.

One of the fundamental engineering challenges of a UI is pulling together all the data. For an e-commerce application, you may want to show catalog items, prices, shipping options, special offers, and other types of information on a single page. You know from Part II of this

book, "Strategic Patterns: Communicating Between Bounded Contexts" that with event-driven applications, this variety of information is stored in multiple, eventually consistent bounded contexts. You also know that these types of systems are share-nothing; in other words, the web application cannot simply query the database of another bounded context because that increases coupling. To solve this problem, you have choices, each with a variety of trade-offs. For instance, you can combine the data on the server or via AJAX calls directly in the web page. Your bounded contexts have the option of returning plain data, usually XML or JSON, or they may return HTML that can be directly dumped onto the page. This chapter has examples of each of these scenarios, along with guidance about when each pattern is relevant and what trade-offs are involved.

Before commencing with the examples, though, this chapter begins by taking you through some of the main UI considerations from high-level decisions—such as which team should own it—to low-level decisions—like which programming language to use. After completing this chapter, you will learn about how the application tier deals with inputs coming from the UI and provides all the infrastructural glue to coordinate actions with bounded contexts.

DESIGN CONSIDERATIONS

It can be quite surprising to see the variety of options and trade-offs that are involved in designing UIs that bring in content from multiple bounded contexts. Some of the options may affect how you design your back-end application programming interfaces (APIs), whereas others may even impact your choice of programming language(s). In fact, your UI could even affect which data needs to be stored by some bounded contexts.

Owned UIs versus Composed UIs

Your first decision when designing a UI is to decide who logically owns it. For instance, it could live within a single bounded context (more specifically a single business component) and be owned by the team responsible for that bounded context. Alternatively, the view could pull in data from multiple bounded contexts but not be owned by any of them.

Autonomous

A UI for an autonomous application belongs to a single business component. It does not need to pull in content from another bounded context. However, this means the business component needs to store locally all the information that should be presented on the UI. To make this possible, the business component has to subscribe to events from other bounded contexts that contain the data it needs and store the data locally. This was discussed previously in Part II and is illustrated in Figure 23-1.

Figure 23-1 illustrates a content-enhancement application that the Catalog bounded context owns. This allows people working in the catalog team to update and override the content for specific products. All the content for products is stored in this business component, so it is fully available. However, when they're updating content, the business staff members want to know how often a product is sold so they can understand how much effort they should put into the quality of content.

This information is retrieved by subscribing to the Sales bounded context's Sale Completed event and stored in the Content Enhancement database, ready to be presented in the autonomous web application's UI.

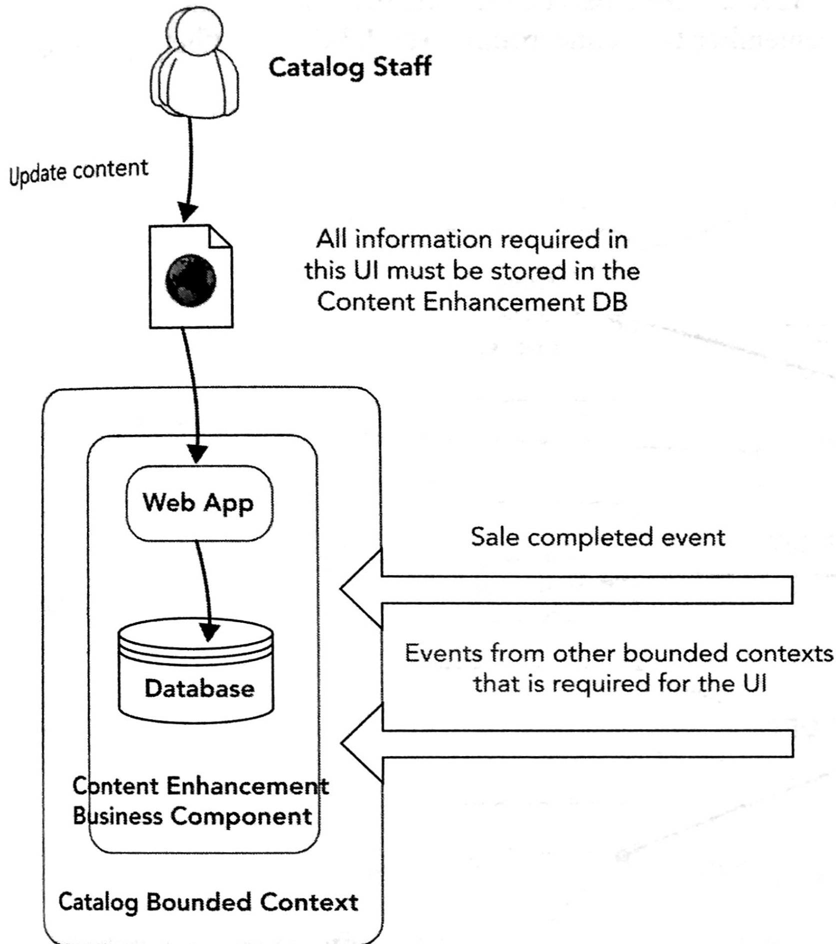

FIGURE 23-1: UI for autonomous applications.

Eventual consistency can be an important consideration with UIs in autonomous applications because the information shown may not be fully up to date. For the example in Figure 23-1, it is fine for the number of sales to be minutes, hours, or even days out of date because the catalog staff just need an idea of an item's popularity. But if data freshness was a big concern, an authoritative application may be a better choice.

Authoritative

When you want the latest snapshot of information from multiple bounded contexts in a single UI, the application has to request each piece of information directly from the authoritative bounded context. You can see this in Figure 23-2.

Figure 23-2 shows an e-commerce web page that calls into multiple bounded contexts, each the authority for the desired information, to get special offers, prices, and other kinds of information. This happens each time the page is requested, so the information is fully up to date, not eventually consistent.

UIs that defer to the authority of each piece of information do not belong to a bounded context. Instead, many companies have dedicated web teams that don't own bounded contexts but are completely responsible for the website. If that approach doesn't work for you, you can let teams that own a bounded context also be responsible for UIs that defer to authority. It's just important to remember that conceptually the UI does not belong to their bounded context.

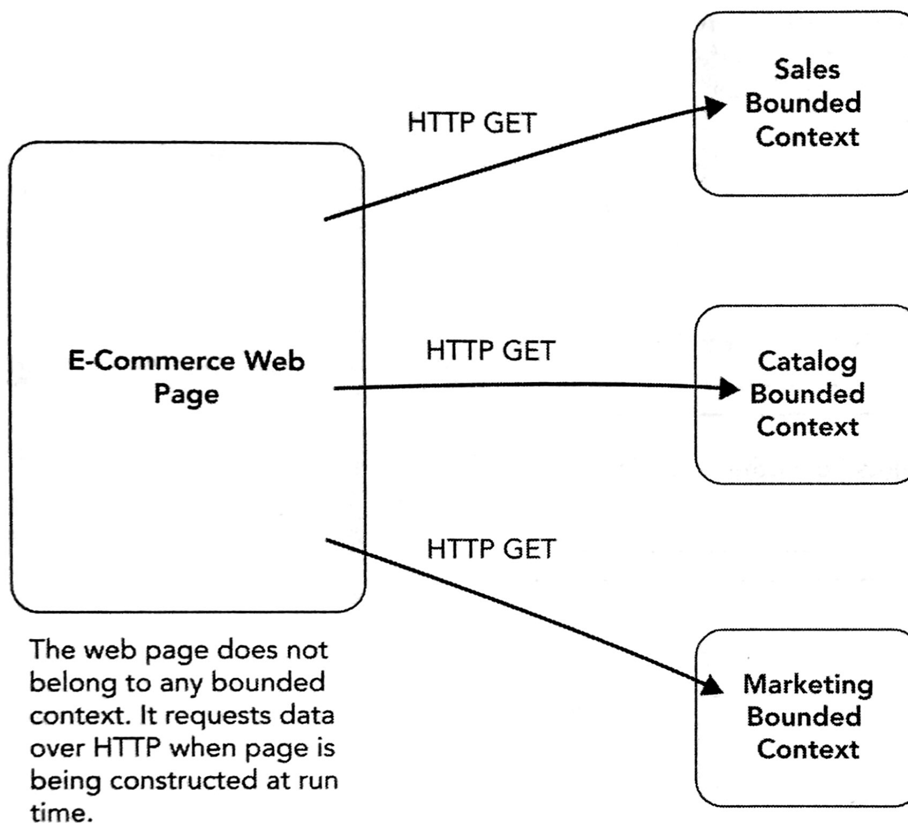

FIGURE 23-2: UI that defers to authority.

Some Help Deciding

A good starting point for choosing between autonomous and authoritative applications is to think about team relationships. If the UI is for a specific department, like an internal tool might be, it may be more efficient to keep it within that team. On the other hand, if the UI forms part of a bigger application that contains many UIs, such as a public website, you may want a dedicated web team to deal with all the web and front-end challenges.

Another consideration is the amount of extra data storage and complexity involved in enabling an autonomous application to have all the data it needs locally. If it is a lot of extra work for a relatively minor use case, the authoritative option might provide the most benefit for the least amount of short- and long-term effort. But if you do need fully up-to-date information, eventually consistent autonomous applications are probably not the best choice.

HTML APIs versus Data APIs

By constructing web pages with snippets of HTML that are returned from each bounded context, you give bounded contexts control of the appearance and behavior of specific regions of a page, as Figure 23-3 shows.

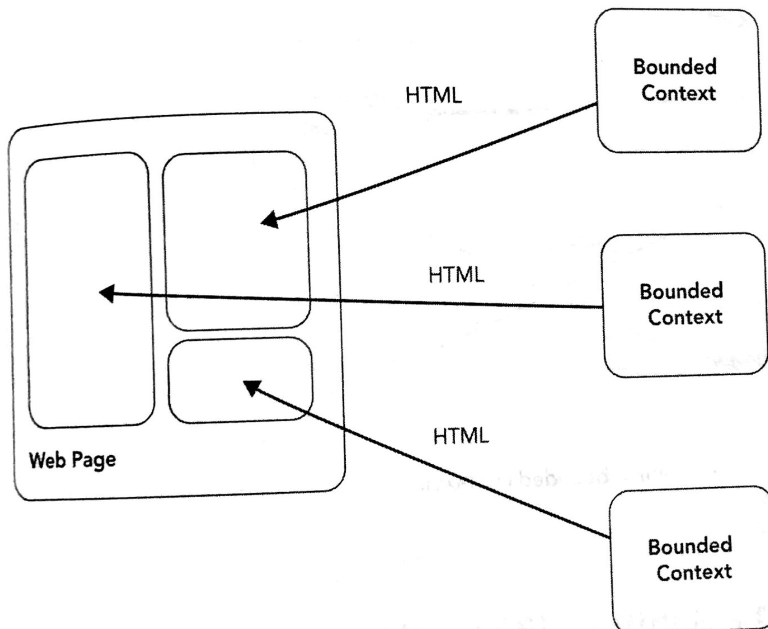

FIGURE 23-3: Composing a web page with HTML provided by bounded contexts.

Another option, which offers less presentational control to bounded contexts, is to have pages only pull in data from bounded contexts. With this alternative approach, you can manage all the presentation concerns in a single location, as shown in Figure 23-4.

Most online experience reports indicate that the second approach is by far the most prominent, and it's usually expressed as JSON APIs. But both approaches can work. One important consideration is whether you provide APIs that are used externally. It was mentioned in Chapter 13, "Integrating Via HTTP with RPC and REST," that dogfooding your API can have a number of benefits in such scenarios.

Client versus Server-Side Aggregation/Coordination

For a UI that pulls in content (data or HTML) from multiple bounded contexts, there is the choice of performing the aggregation on the client or the server. By making each request an AJAX request inside the web page, you can avoid the complexity and additional failure point of the server-side application. Conversely, you will have more complexity on the client as JavaScript. Building Single Page Applications (SPAs), as many teams are now, is one case in which this is less of a problem. The general recommendation on this topic tends to favor the client, but both approaches are in wide use. Figure 23-5 illustrates knitting together content on the client, whereas Figure 23-6 illustrates the server-side approach.

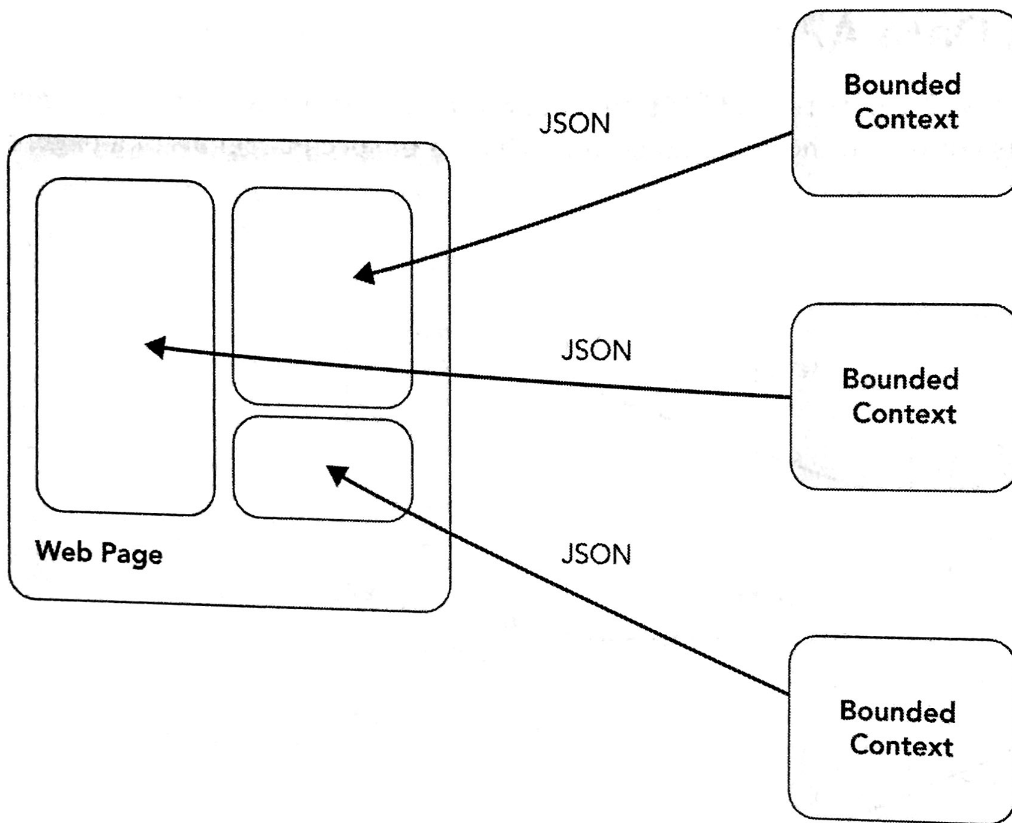

FIGURE 23-4: Pulling in data from multiple bounded contexts.

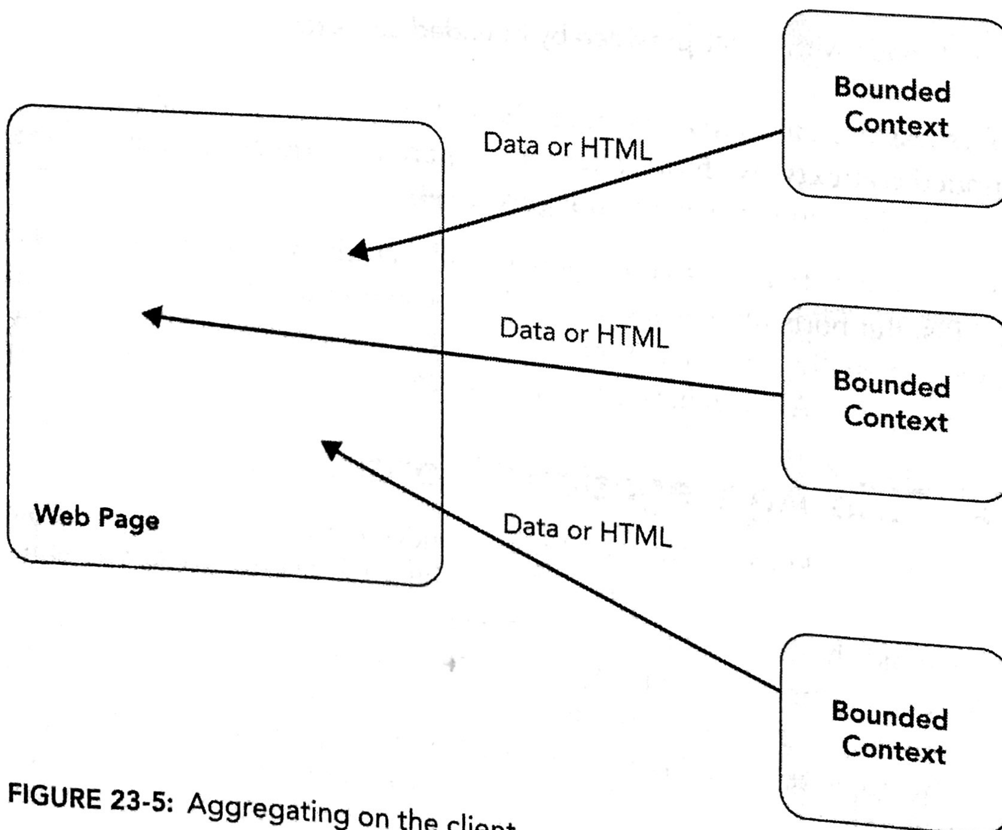

FIGURE 23-5: Aggregating on the client.

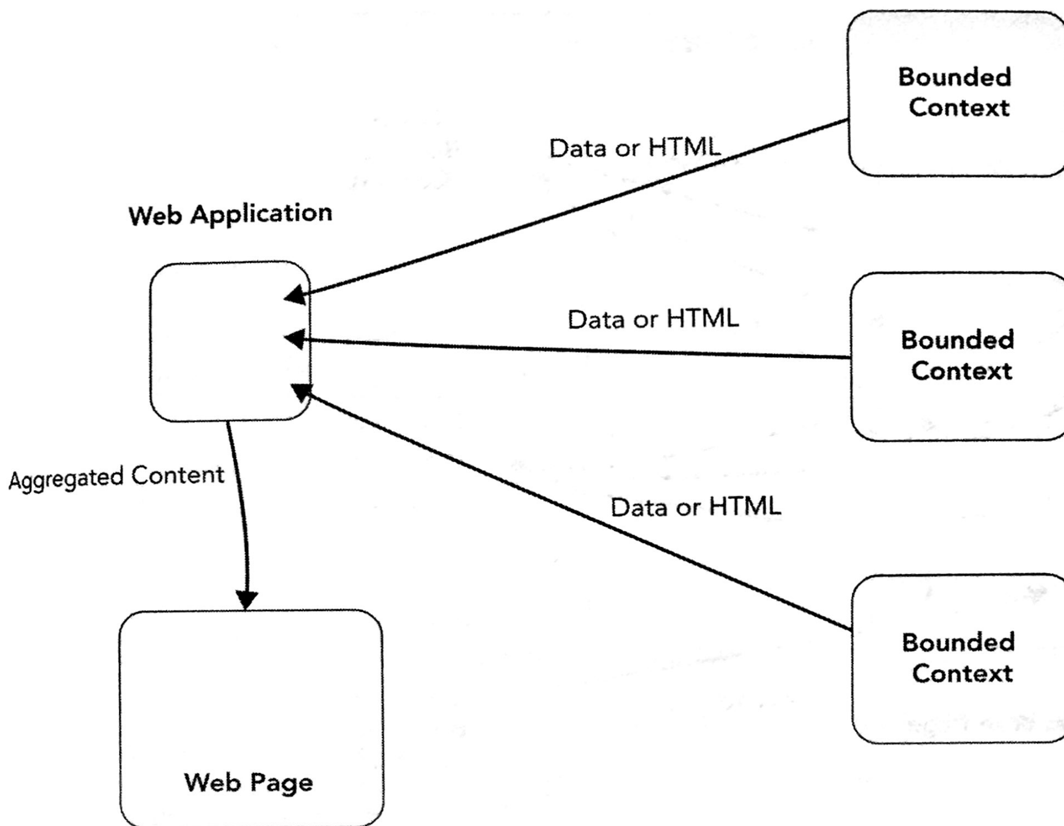

FIGURE 23-6: Aggregating on the server.

EXAMPLE 1: AN HTML API-BASED, SERVER-SIDE UI FOR NONDISTRIBUTED BOUNDED CONTEXTS

Even when all your bounded contexts live inside the same solution and run as a single application, UI composition can still be useful for partitioning presentational responsibility among bounded contexts. When one bounded context would like to alter its portion(s) of a page, the changes may be confined to that bounded context, meaning no interference with others. This is the same intention that motivates the Single Responsibility Principle (SRP). In this section, you implement this scenario using ASP.NET MVC's `RenderAction()`. You're going to create a simple page that pulls in HTML content from three bounded contexts that live inside the same solution, as shown in Figure 23-7.

To begin this example, you need to create a new ASP.NET web application called `PPPDDD.NonDist.UIComp`. Choose the Empty template, and check the MVC check box. This application contains only a single view, which will be the composite UI, so it's fine for it to be the initial page of the application. To achieve this, add a class called `HomeController` in the `Controllers` folder with the content shown in Listing 23-1.

> **NOTE** *ASP.NET MVC's default route is to look for an* `Index()` *method on a controller called* `HomeController`. *If your project has one of these, it is used to respond to the base URL (/).*