In the first section, we deliberately split the traditional CRUD (create, read, update, delete) into separate models for commands. The separate models make changes to data (create, update, delete) and to the other operations that query (search/read) data, to enable them to be changed and scaled independently. However, they were still synchronous in nature, dependent on the datastore's performance and availability.

We can make use of this separation and independently refactor the change operations to be asynchronous, without touching the query path. In our case, we do this by introducing IBM MQ instead of HTTP as the mechanism for the update. Just to be clear what we mean by this, we are not talking about changing only the transport from HTTP to IBM MQ. We are also changing the interaction pattern from request/response to fire and forget. This way, after a request has been made to change data, we can respond immediately to the calling system that the request has been received. We do not have to wait for it to be completed. So, we are no longer dependent on the back-end systems availability or performance. IBM MQ's assured delivery means that we can be confident that it will eventually happen. Furthermore, we can throttle and control when the updates are applied, so that in busy periods they do not affect the performance of queries.

Clearly this model introduces challenges. We don't know exactly when the update will occur. And there might be other updates from other consumers, too. So, we can never be entirely sure of the status of the data in the back-end system. Nowadays we use the term *eventual consistency* to describe this situation. Clearly it is better suited to some business scenarios than others. In our example, we decided that the increased availability and response time on updates to our "product" data, and the potentially more consistent performance on queries, are more important than knowing that the data is 100% consistent all the time.

The *CQRS* (Command Query Responsibility Segregation) pattern has become popular in recent years. Data changes (commands) and reads (queries) are treated as separate implementations, to improve reliability and performance. The integrations for these two halves were already separate, but they were both acting synchronously on the same data source. What we are doing in this section can be described as implementing the "command" part of this pattern. In other words, we change the synchronous data changes into a series of asynchronous commands. In later sections, we look at creating even more separation on the "query" side.

Figure 6-60 on page 209 illustrates this pattern.