Microservices based Smart Shopping

4DV609 - Architectures for Service-based Systems
Project Development

Submitted By:

Weibin Yu
Sloane Petit
Sarpreet Singh
Richard Randak
Prasannjeet Singh
Amir Mirzaeian

Table of Contents

1. Introduction

Intuitively, monolithic applications are easy to build and understand, however, in this day and age when every industry predominantly focuses in scaling their systems in a large scale, monolithic applications start creating problems. There is no denying the fact that these applications have their own advantages such as reduced communication costs, technological consistency and comparatively less effort; however, problems like lack of scalability, tight coupling have forced many organizations to shift their applications from monolith to a more flexible – Domain Driven Design, or DDD.

As mentioned, DDD is a flexible architecture where the whole system, also called as a domain, is decomposed into smaller subdomains. A new application can be developed via the Domain Driven Design by identifying the Subdomains via analysing the business and identifying the different areas of expertise. It easily solves the problem of scalability as only those subdomains can be scaled which are used (or called) more often, while down-scaling other subdomains which are rarely used. Later, one or more microservices are derived from each Subdomain. Microservices run independently and can be developed using different technologies on different systems.

To demonstrate the domain-driven design, we have developed Microservices for a Smart-Shopping application where a user compares prices of different products in multiple stores and sort them via price or by distance of the store from the user-location.

2. Running the Program

Here we describe how to run the SmartShopping application in a Windows Platform. The instructions should be the same for computers from other platforms. Note that the DOCKER_HOST_IP should be set before continuing with the following commands.

1. Download the repository from GitHub with the following command

```
git clone
https://github.com/prasannjeet/SmartShopping.git
```

2. From inside the project folder, run the following Maven command to compile the project and create .jar files

```
./mvnw clean package
```

3. Now run the following command to build docker services

```
docker-compose build
```

4. Finally, run the following command to start all the docker-containers

```
docker-compose up
```

5. The project should be up and running now. All the commands can be run in the following URL:

```
http://{DOCKER_HOST_IP}:7088/
```

3. Domain-Driven Design

Based on the assignment document, let us list down all the functional requirements of the application:

- Provide list of all the products in the system on user's request.
- Edit items from a user's shopping list on their request.
- Add a product, its price and stores where it is located on user's request.
- Update a product's price and/or it's store location on user's request.
- Sort products in a shopping list according to the price or store location.
- Provide nearby stores based on the user's GPS location.
- Provide a single product's price in different stores.
- Scrape store-website automatically to collect new products and update existing products.

In this application, the whole end to end process encompasses one domain. We call it the Smart-Shopping domain. Further, based on the areas of expertise, we have divided our domain into four different Subdomains which are as follows (Figure 3-1):

1. Store Subdomain
2. Product Subdomain
3. Cart Subdomain
4. User Subdomain

Let us now briefly go through each subdomain:

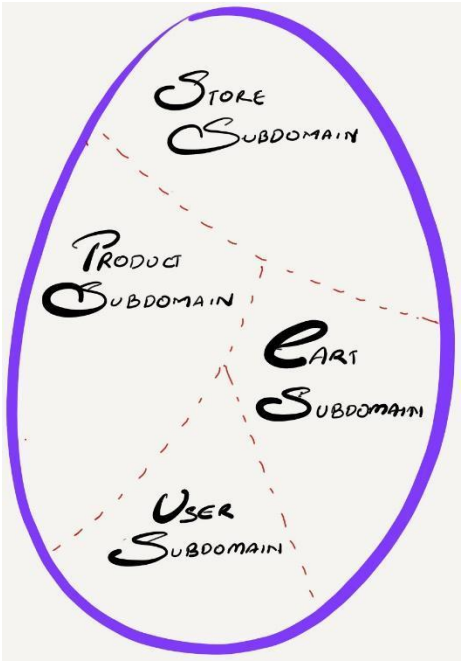

Figure 3-1: Identified Subdomains

3.1 Store Subdomain

As the name suggests, the store domain is responsible for managing all the services related to the store. It has all the information of the available stores in the application including its location, the products stockpiled at each store and the price of a particular product in all the

stores. Apart from this, the store subdomain is also responsible for scraping various store websites to look for new products, availability of products and updated price, if any. This is subdomain also responsible for finding out the nearest stores based on the location of the user.

3.2 Product Subdomain

Similarly, the product subdomain is basically a global repository of set of all the available products in all the stores in the scope. Functions like Creating, Reading, Updating and Deleting (CRUD) products are done within the product subdomain.

3.3 Cart Subdomain

This subdomain contains all the information about every user's shopping list, and is responsible for all the CRUD operations in the list. Moreover, it is also accountable for the most important business logic of the application, which is sorting the products of a user's shopping list according to the price or their GPS location.

3.4 User Subdomain

The user subdomain contains the repository of all the registered user in the application. It is responsible all the CRUD operations for all the users.

3.5 Design Issue

Originally, few members of the team were under the impression that functions such as Web-Scraping as well as GPS location finding should be allocated in their own subdomain. However, it was, at a later stage, realized that Web-Scraping is a very small function which can be harnessed using a third party off the shelf library, and can be part of the Store Subdomain. Furthermore, the GPS functionality was scraped off altogether, as the location of all the stores are fixed and stored in the Store Subdomain, and the user-location will be provided by the client.

4. Derived Services

Derived from the above identified subdomains are the services, as shown in the Figure 4-1 below:

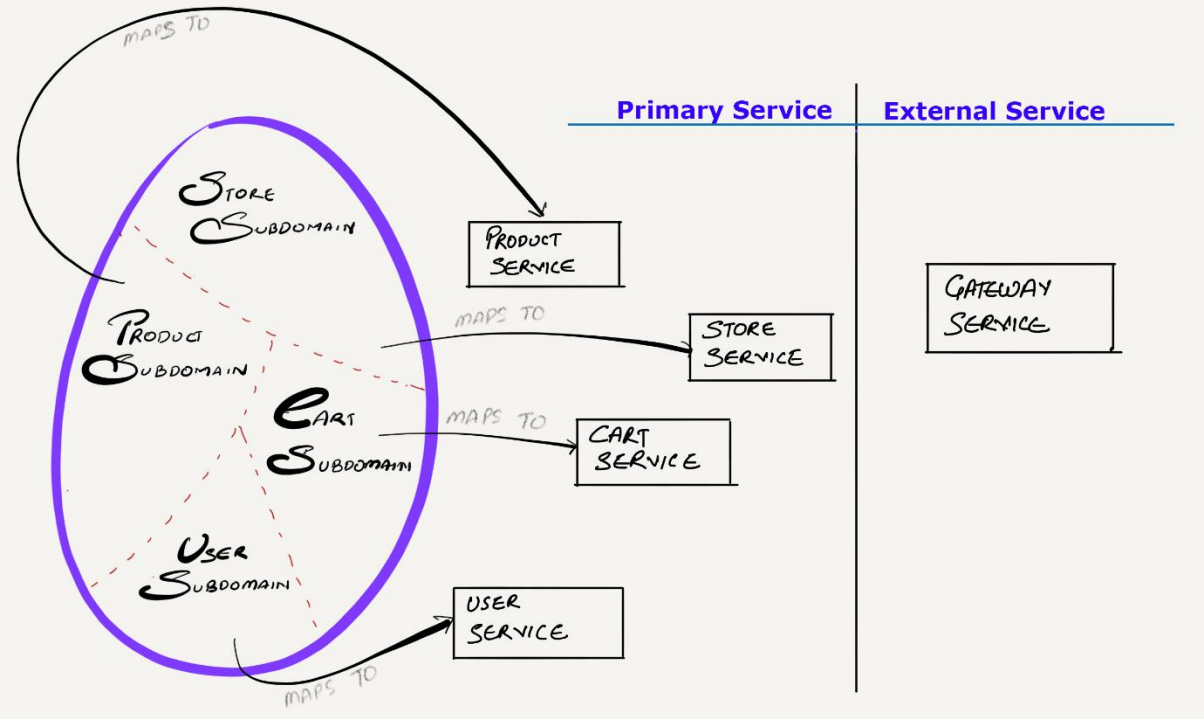

Figure 4-1: Identified Services

List of identified services are:

5. Store Service
6. Product Service
7. Cart Service
8. User Service
9. Gateway Service

In this application, every domain corresponds to one service and in addition, we also have an external Gateway Service which is responsible for relaying commands from the client side to the application and vice-versa. As can be seen in Figure 0-1, the User, Cart, Store and Product service are all instantiated inside a Docker Container. These services interact with each other via evets, which is handled by Eventuate. On the other hand, the gateway service interacts with the client and the application using HTTP. It is also built using the Spring Framework. Other nitty-gritty details regarding each individual service will provided in their respective reports.

5. Service APIs

Let us briefly have a look at the API's of each service:

5.1 Store Service

Initialize a Store

- **URL:** `/stores/init`
- **Method:** `POST`
- **Headers:** `Content-Type: application/json`
- **Body**

```
{
    "name": "willys",
    "website": "/json/websiteExample.json",
    "location": "12.2"
}
```

- **Response**

```
{
    "id": "0000016703697ed2-0242ac1100060000",
    "name": "willys",
    "website": "/json/websiteExample.json",
    "location": "12.2"
}
```

Add Product to Store

- **URL:** `/stores/{storeId}/product`
- **Method:** `POST`
- **Headers:** `Content-Type: application/json`
- **Body**

```
{
    "name" : "mozzarella cheese",
    "barcode" : "101011201",
    "price" : "56",
    "hasWeight" : true
}
```

- **Response**

```
{
    "id": "000001670372b9b2-0242ac1100060001",
    "name": "mozzarella cheese",
    "price": "56",
    "barcode": "101011201",
    "hasWeight": true
}
```

Update Product Price

- **URL:** `/stores/{storeId}/product`

- **Method:** PUT
- **Headers:** Content-Type: application/json
- **Body**

```
{
    "price": "59",
    "barcode": "101011201"
}
```

Start Web Scraper

5.2 Product Service

1. Get Products
2. Get Product

5.3 Cart Service

1. Add product
2. Update product quantity
3. Delete product
4. Get a cart
5. Get carts
6. Sort cart by stores distance
7. Sort cart by products price

5.4 User Service

Register a User

- **URL:** /users
- **Method:** POST
- **Headers:** Content-Type: application/json
- **Body**

```
{
    “username”: “bob”,
    “password”: “pas111”
}
```

- **Response**

```
{
    “id”: “ksnc123n212”,
    “username”: “bob”,
    “password”: “pas111”
}
```

Retrieve Users

- **URL:** /users
- **Method:** GET

- **Response**

```
[
      {
          "id": "ksnc123n212"
          "username": "bob",
          "password": "pas111"
      },
      ...
]
```

Retrieve a User

- **URL:** `/users/{userId}`
- **Method:** `GET`
- **Response**

```
{
      “id”: “ksnc123n212”,
      “username”: “bob”,
      “password”: “pas111”
}
```

Delete a User

- **URL:** `/users/{userId}`
- **Method:** `DELETE`
- **Response**

```
{
      “id”: “ksnc123n212”,
      “username”: “bob”,
      “password”: “pas111”
}
```

6. Collaboration

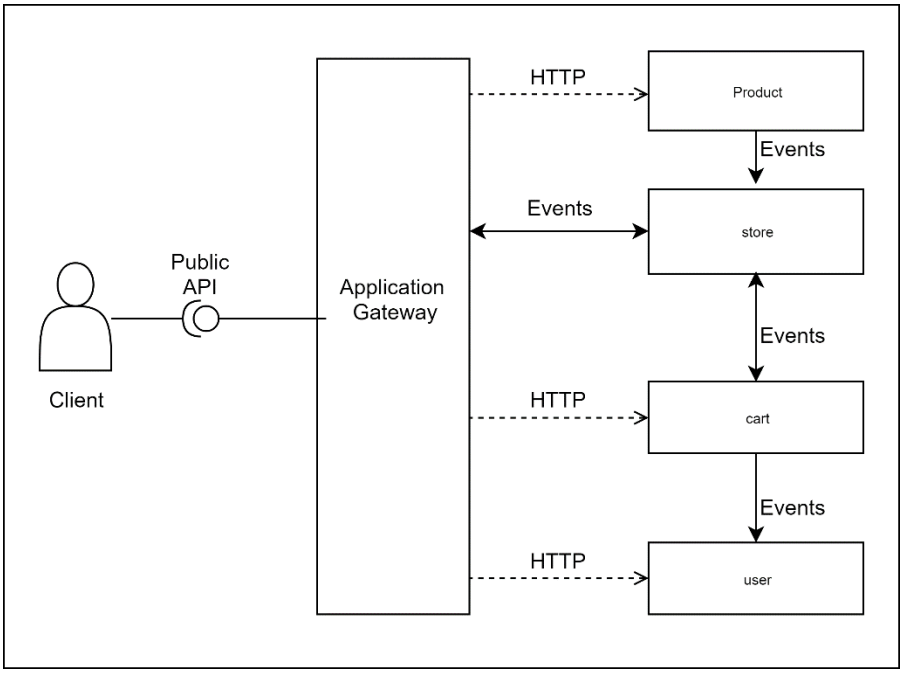

Figure 6-1: Collaboration between Services

As it can be observed from the image above, client side only interacts with the gateway for all the public operations. In turn, the gateway redirects these request to the respective services. Other services also interact with each other in a few occasions. All these interactions will be comprehensively discussed in the following sections.

7. Design detail

7.1 Microservice Composition Design

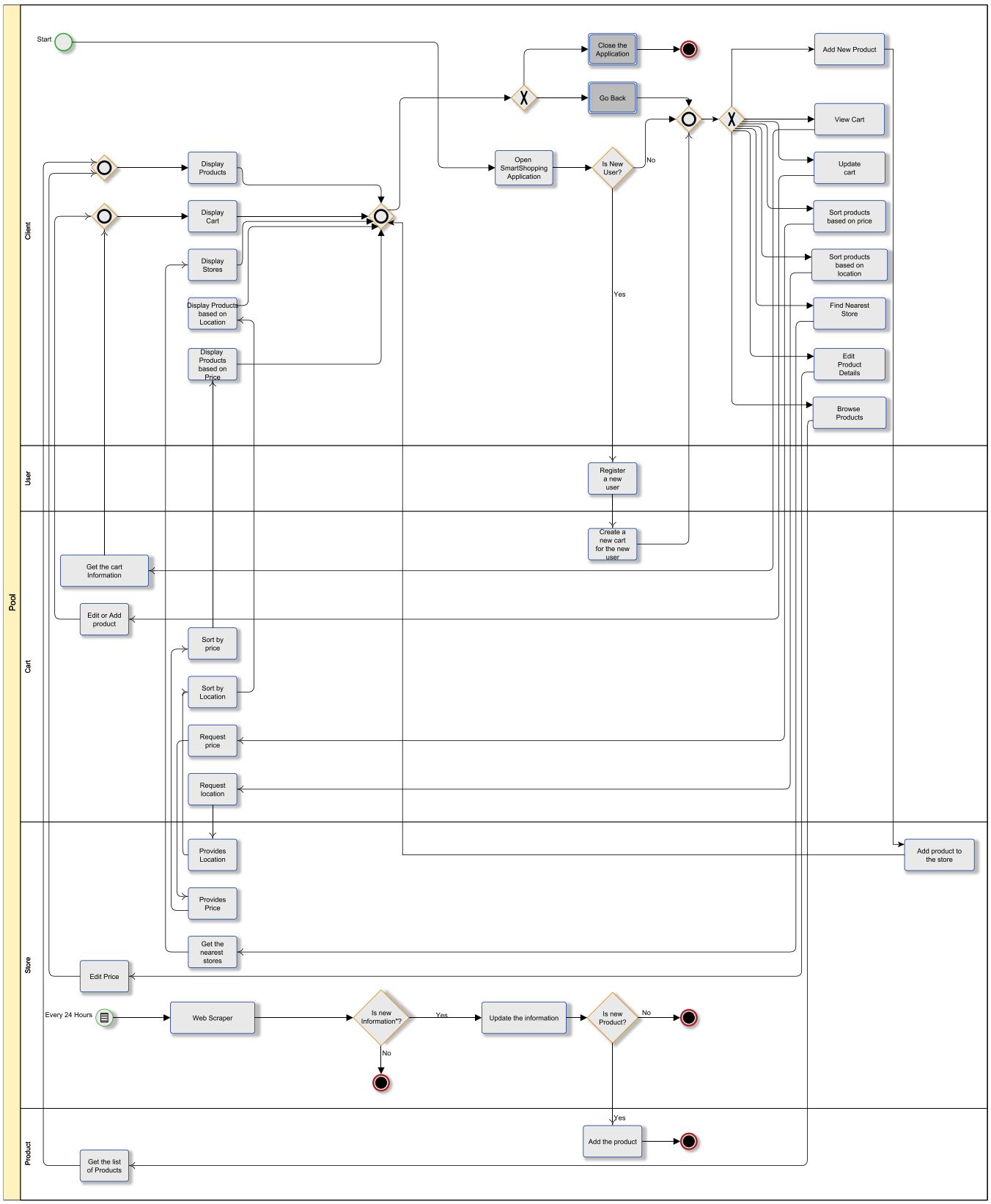

Figure 7-1: Business Process Model and Notation for Smart Shopping

The microservice composition is described using BPMN above. It should be noted that this section aims to discuss the high-level interactions between services. Therefore, the protocols used for these interactions (like HTTP and Events)are not explained here since they have been discussed in the “Event-Driven Architecture” section.

As it can be seen from BPMN, when a user starts the application for the first time, the *user* service of the application detects it and registers the user. Consequently, the user service sends a request to the *cart* service to create a new shopping cart for the newly registered user. Therefore, the user can access the menu. On the other hand, if the user has been already registered, then the user can directly access the menu.

The user can opt for following options which are described briefly below:

1. **Browse Products:** When the user selects “Browse Products” from the menu, the *Product* service sends the list of all the products to the *Client*.

2. **View Cart:** As soon as the user selects “View Cart”, the *Cart* service sends the user’s cart to *Client* to be displayed.

3. **Update Cart:** Selecting “Edit Cart”, the user can edit the existing cart. User can also add a new product to the cart while browsing products.

4. **Sort Products based on Price:** After selecting this option, *Cart* service requests the price form *Store* service due to the fact that price is stored in *Store* subdomain. The price is sent back to Cart where is responsible for sorting the products by price. Finally, the sorted list of products will be sent to *Client* in order to be displayed.

5. **Sort Products based on Location:** This option has nearly the same functionality as option 3. The location is stored in *Store* subdomain, so a query will be sent from *Cart* to *Store* to provide location. After the provision of Location by Store, Cart service is able to sort the products based on location and send a command to display the sorted products on the *Client*.

6. **Find the nearest store:** Since the Stores’ locations are stored in *Store* subdomain, when the user selects “Find the nearest store”, a request which contains the user’s location will be sent to *Store* service. All the nearby store instances reply to this request with their distance and name and the nearest stores will be finally sent to the *Client*.

7. **Edit Products' Details:** The user is able to change the products' price which is stored in *Store* subdomain. Therefore, when the user tries to change the price of a product in a particular store, the *Store* service make changes and then the product with updated price will be sent to the *Client*.

8. **Add new product:** The user is also allowed to add a new product to the product repository of the application. When the user tries to add a new product to a store, the product is added to that instance of the store where the user wished to add it, after which it requests the product service to add the same product to its repository.

After displaying each of abovementioned option's output on the Client side, the user is able to close the application or request a new option.

Web Scraper: Web Scraper is located in *Store* subdomain and has been exploited to update the products' price from the stores' websites within a pre-defined intervals (every 24 hours in our case). If the products' information needs updating, the *Store* service does the updating.

If through web scraping process, a new product is found, the *Store* service adds the product to its domain and also sends a request to *Product* service to create the product there as well.

7.1.1 Composition

As it can be seen from SmartShopping architecture, the services are composed in *Choreography* owing to the fact that in our design, the decision logic is distributed with no centralized point. Each service in our system acts independently, knows when to execute and with whom to interact. In fact, all the entities communicate with no central conductor or overall responsibility. In contrast, each service carries a part of responsibility on its own.

7.1.1.1 Advantages of Choreography in SmartShopping:

1. Different responsibilities are placed in different services. Hence, each service has its own independent code repository.
2. Dependencies between services is minimized which results in less complexity.
3. Every service subscribes to the events that it is interested in and contains its own logic which leads to lose coupling.

7.2 Event-Driven Architecture

The following diagram describes the event driven architecture of the system

Let us discuss each interaction individually:

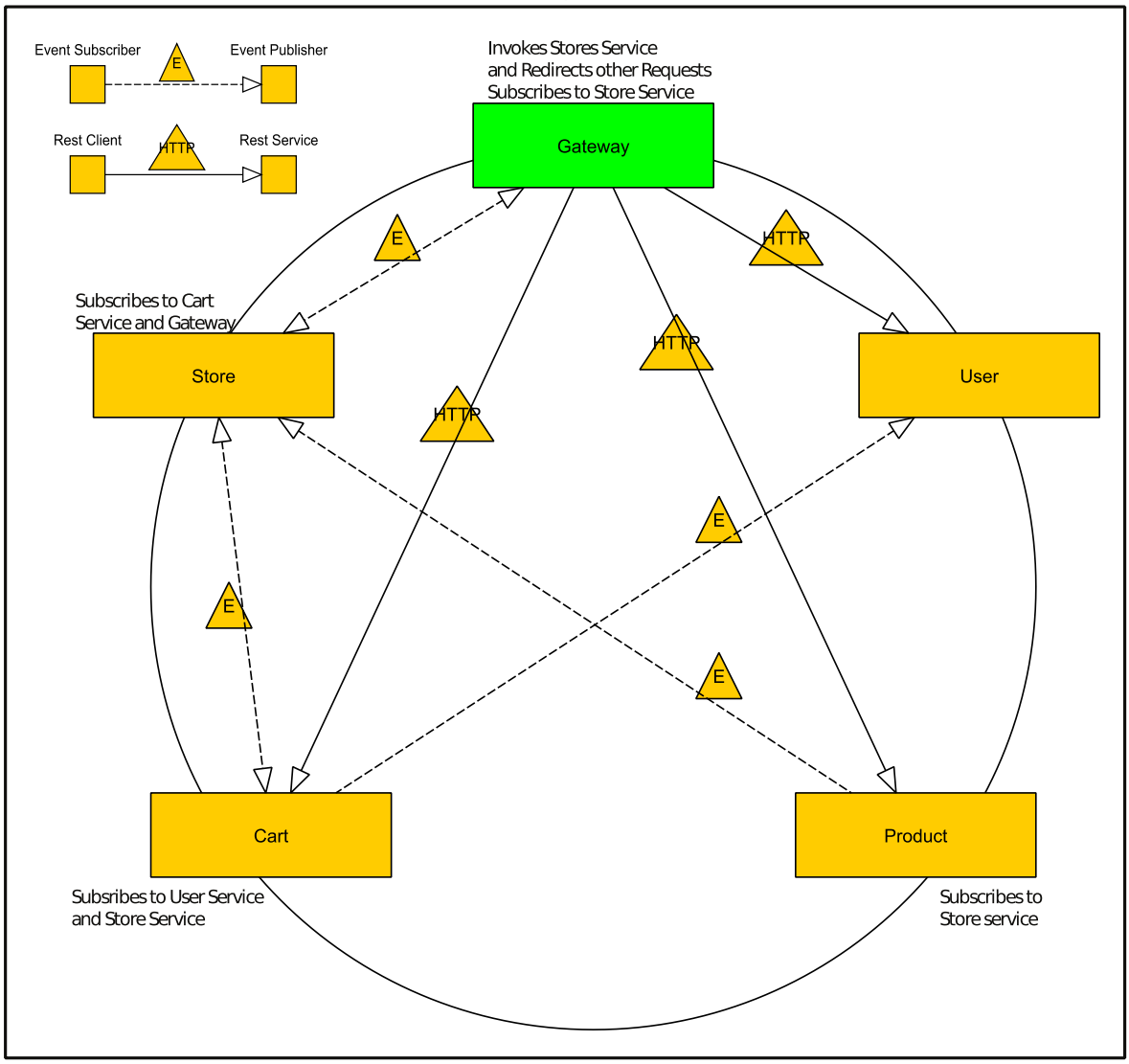

Figure 7-2: Event Driven Architecture for Smart Shopping

- **Gateway-User:** The gateway interacts with the User-Service when a new user tries to use the application for the first time. It automatically invokes the User API to create a new user inside the User-Service. This interaction is done via HTTP.
- **Gateway-Cart:** The gateway interacts with the Cart Service via HTTP for retrieving or modifying the shopping cart of a user on a their request.
- **Gateway-Store:** As the store service is instantiated for every new store, gateway interacts with the service via Events. Whenever a user requests for stores near their GPS location, the gateway publishes an event which is subscribed by all instantiated store services. Meanwhile all the stores that are near to the GPS location publish another event which, in turn, is subscribed by the gateway for receiving the list of all the nearby stores. This list is then sent to the user. Moreover, instantiation of a new store also happens here.
- **Gateway-Product:** The gateway interacts with the Product-Service via HTTP to retrieve the list of all the available products in the application, which is stored in the Product Service.
- **Store-Product:** Whenever a new product is added to the store, or any existing product is modified/deleted, either via a user-request or via the web-scraper, the store service(s) publish an event which is subscribed by the Product-Service so that it can update its product-repository.
- **Store-Cart:** As the cart is responsible for sorting all the products in the shopping list according to the store locations, it interacts with all the instances of store services via Events in the same fashion as the gateway interacts with the store services to get the list of nearest store locations.
- **User-Cart:** Every time the User-Service creates a new user, it publishes an event which is subscribed by the cart service for creating a new shopping cart for the respective user.

7.3 Event Driven Architecture Design

As it is illustrated in Figure 6-1, when a service creates or updates an aggregate in the database, it publishes an event. Hence, updating the database and publishing an event needs to be atomical. However, it is possible that a service crashes after updating the database but before publishing an event which results in an inconsistent state in the system.

In order to solve this problem, SmartShopping benefits from event sourcing technique. In fact, event sourcing is a practical way to update states atomically and publish events. In event sourcing, instead of focusing on current state, we need to concentrate on the changes that have taken place over time. It can also be considered as modelling our system as a sequence of events. Consequently, when a service which uses event sourcing approach creates or updates an aggregate, the service saves one or more events in the database. It reconstructs the

current state of an aggregate by loading the events and replaying them. Owing to the fact that in our system events are the state, we do not suffer from having the problem of atomically updating state and publishing events.

The aforementioned concept can be exemplified by the SmatShopping's sub domain- cart's life cycle -which can be modelled as the following sequence of events:

Events
1. Shopping Cart Created
2. Item Added To Cart
3. Item Quantity Updated
4. Item Removed From Cart

The idea here is to represent every *application's* state transition in a form of an immutable event. Events are then stored in a long as they occur (called "event store").

Advantages of event sourcing technique:

- Ability to put the system in any prior state which can be useful for debugging.
- Having a history of the system which gives more benefits such as audit and traceability.
- No need to use a fancy database to store our events. A standard MySql table can be used as events storage. Furthermore, it is easy to change database implementation due to the ephemeral nature of event sourced data structure (since all state is derived from events, we are no longer bound by the current state of our application). Hence, there is a possibility to switch to a better tool at any point due to an incredible amount of freedom that event sourcing provides.
- The kind of operations made on an event store is very limited, making the persistence very predictable and thus easing testing.
- Ability to implement temporal queries that determine the state of an entity at any point in time.

As it can be seen in Figure 6-1, there is a platform called Eventuate which is responsible to store events. This event store can be considered both as database and event broker owing to the ability to keep the sequence of events and have an API for subscribing to events.

Eventuate is a platform for developing asynchronous microservices. The platform consists of two products:

- **Eventuate Tram** - a framework for traditional JPA/JDBC-based microservices. You can easily add Eventuate Tram to your Spring framework-based microservices without having to rewrite your business logic.
- **Eventuate ES** - a microservices framework that implements an event sourcing-based programming and persistence model.

7.3.1 CQRS pattern

In microservice architecture we cannot join the database tables from different service since each table is owned by a different service and is only accessible within that service. As a result, the traditional queries that join the tables owned by multiples services are not practical anymore.

SmartShopping has exploited the Command Query Responsibility Segregation pattern so as to deal with the implementation of query operations. In this pattern, we have divided the application into two parts: the command-side and the query-side. This means that the data models used for querying and updates are different.

In fact, on the read-side, only query methods are available, while on the write-side, only update (or command) methods are available.

The command-side handles create, update, and delete requests and emits events where data changes. However, the query-side handles queries by executing them against one or more materialized views that are kept up to date by subscribing to the stream of events emitted when data changes.

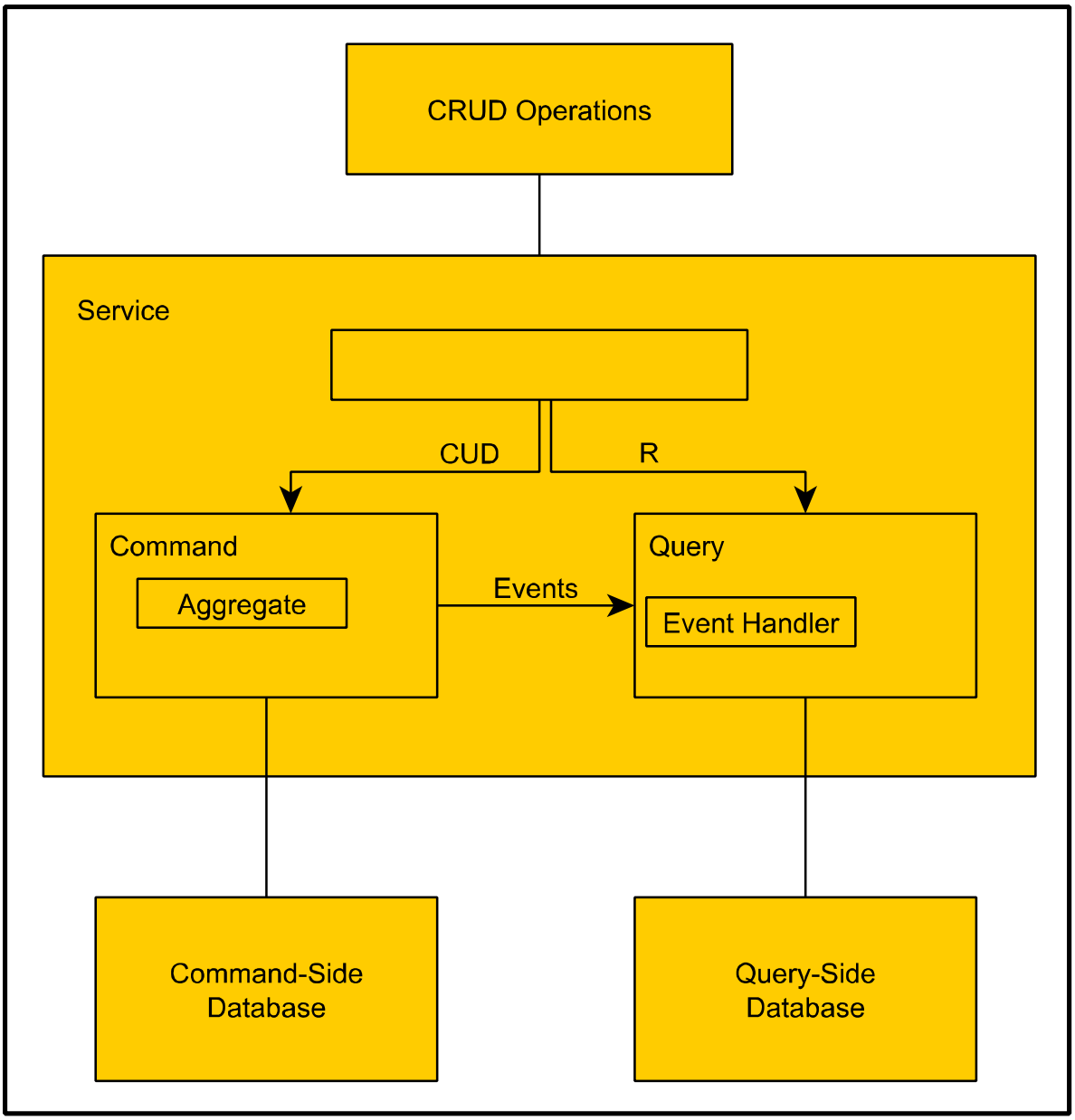

Figure 7-3: Command-Side vs Query-Side Architecture

Having this separation enables us to dive into how to divide our business logic into commands, events, and queries. *Commands* are requests for the system to perform an action. A sample command may be “create a user” or “ remove an item from the cart”. *Events* are notifications that something has happened. Events can be processed multiple times by multiple consumers. A sample event is “ a new user has been created”. Lastly, *queries*, are requests for information. A sample query can be “show me a specific user”.

Advantages of CQRS pattern:

- CQRS makes it possible to implement queries in a microservices architecture, especially one that uses event sourcing.

- The separation that takes place in CQRS pattern often simplifies the command and query sides of the application.

7.4 Other Design Issues

- A major design issue we faced was in deciding the database structure for the services. Teams had a conflicting view on which service should be responsible for storing the product price. While in first look, it might seem that the price should be stored in the product service, a deeper observation and the fact that each store may have different prices for same products led us to the conclusion that the prices should be stored in the *store* service.
- Other major change to the design was made in the checkpoint meet. Initially store service was supposed to be a single instance containing two databases. One that saved all the stores, and the other that saved all the products with prices in each existing stores. This definitely would have caused problems while scaling as we might have had multiple copies of same products for saving price in different stores. The design was later changed so that each store has it's own instance of the store service. This way individual stores could be scaled in case they become bigger and popular in future, causing less overhead.