



ADVANCED DATABASE SYSTEMS
2023/2024

Command Query Responsibility Segregation

Samuel Janek (xjanek03)

Pavel Šesták (xsesta07)

Brno, December 3, 2023

Contents

1	Introduction	2
2	Specification	2
2.1	Custom assignment	2
2.2	System sequence diagram	3
2.3	Commands	4
2.4	Queries	4
3	Design	8
3.1	High-level Component diagram	8
3.2	Detail Component diagram	9
3.3	Synchronising data stores	9
3.4	E-R Diagram	10
3.5	MongoDB	11
3.5.1	Data validity period	11
3.5.2	Scalability	11
3.5.3	Definition of collections	11
4	Technologies	15
5	Deployment	15
6	Testing	15
7	Conclusion	17

List of Figures

1	System sequence diagram	3
2	High-level Component diagram	8
3	Detail Component diagram	9
4	ER - Diagram	10
5	Deployment Diagram	15
6	Result of automatic tests	16
7	OpenAPI Specification Write Service	16
8	OpenAPI Specification Read Service	17
9	Postman workspace	17

List of Tables

1 Introduction

Command Query Responsibility Segregation is a general architectural pattern that separates the two main responsibilities in a system: commands (Commands) and queries (Queries).

Command service

On this side, the requests for writing and editing data are processed. The complete business logic of the application, including input data validation, occurs here. To keep the application data consistent, the data is written to relational database that allows transactional processing and meets ACID properties.

Query service

This side of the application is used to query the data, that are stored here in a more optimal form for reading than in the relational database. Data is typically reordered and denormalized to eliminate JOIN operations and speed up querying. Thus, NoSQL databases are typically used. The synchronization of these NoSQL databases is based on events that are retrieved from the Command service. Typically, there is some delay between writing data to the relational database and writing it to the NoSQL database.

2 Specification

In this section, we will briefly introduce the project and discuss the operations in our system. Since this is an e-shop, we will assume that the frequency of reading data greatly exceeds the frequency of writing and modification.

2.1 Custom assignment

In the context of the assignment, let us consider a virtual marketplace similar to the portals Amazon or Aliexpress. Within the portal, both vendors and customers can register. Customer can display the goods offered by vendors. The vendors are grouped into categories according to the products they currently offer on the e-shop. The customer can browse the marketplace and add goods to the cart. He can then order the goods and create a review on the items he has ordered. The review can be viewed on the item itself and on the vendor's profile.

2.2 System sequence diagram

This diagram models the most common scenario of the customer’s use of the system.

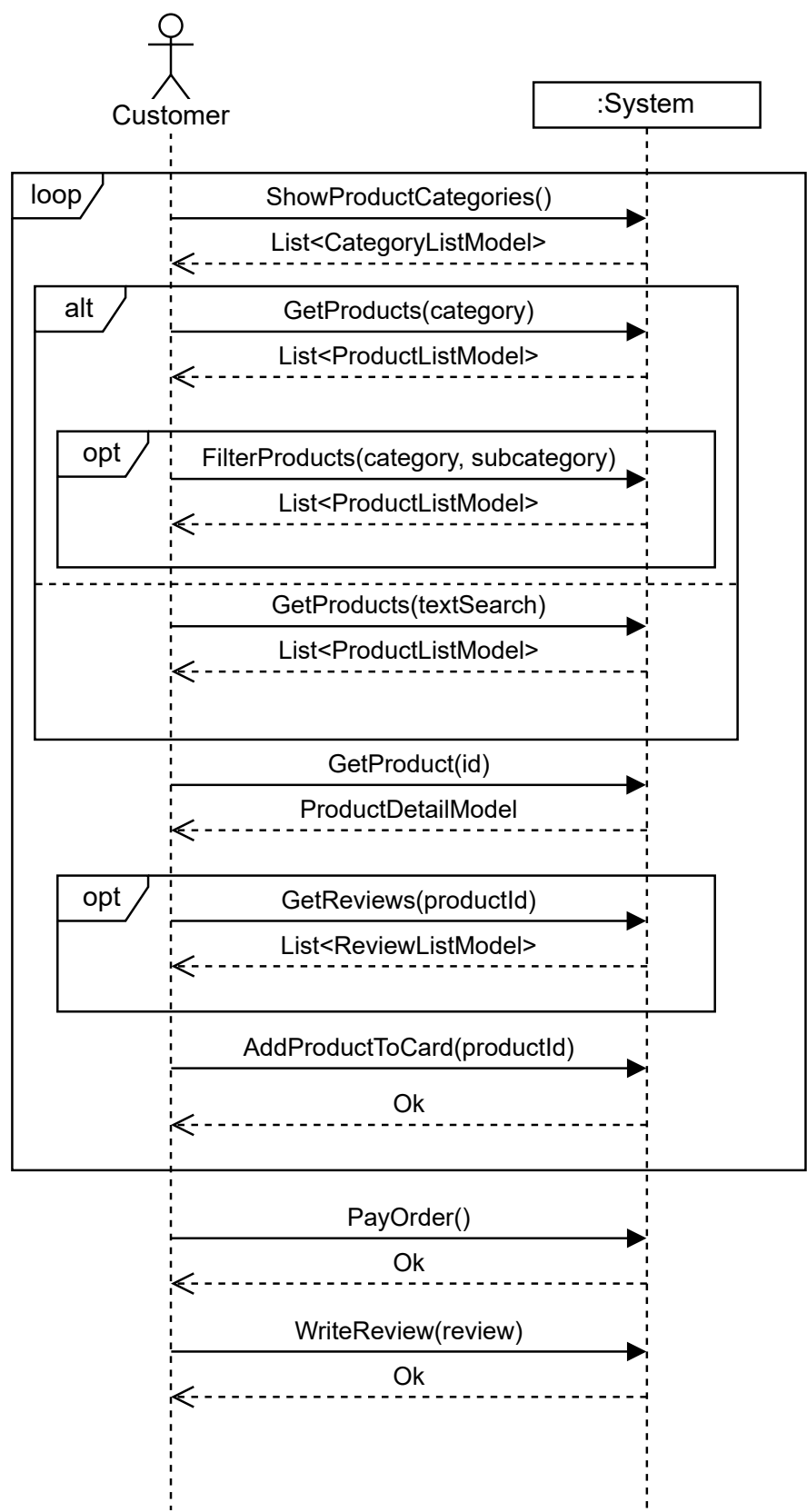


Figure 1: System sequence diagram

2.3 Commands

In this section we describe the commands over each identified entity within the system.

Vendor

- Create
- Update
- SoftDelete

Customer

- Create
- Update
- Anonymize (We can't delete the customer due to their order history, but we must remove their GDPR-sensitive data.)

Product

- Create
- SoftDelete

Order

- Create
- AddProductToCart
- Pay

Review

- Create

Address

- Create
- Update
- Delete

2.4 Queries

In this section, we describe queries used in our system. Within the structure, we specify the expected use of the endpoint and the data structure in JSON format, where square brackets are used to denote collections and curly brackets for objects. The attribute is in the format `<name:datatype>`. Most queries are available to everyone, including unregistered users, as forcing registration to view products could negatively affect conversion rates (A measure of how many visitors become clients).

Get customers

Usage: Admin panel

Role: Manager, Administrator

```
[
  {
    id: bigint,
    fullName: string,
    email: string,
    phoneNumber: string
  }
]
```

Get vendors

Usage: Admin panel

Role: Manager, Administrator

```
[
  {
    id: bigint,
    name: string
  }
]
```

Get customer detail by email

Usage: Customer profile page

Role: User (can see only his/her own account details), Manager, Administrator

```
[
  {
    id: email as string,
    firstName: string,
    lastName: string
    phoneNumber: string,
    addresses: [
      {
        country: string,
        zipCode: string,
        city: string,
        street: string,
        houseNumber: string
      }
    ]
  }
]
```

Get vendor detail by id

Usage: Vendor profile page

```
[
  {
    id: bigint,
    name: string,
    category: [string],
    address: {
      country: string,
      zipCode: string,
      city: string,
      street: string,
      houseNumber: string
    }
  }
]
```

Get products

Usage: Landing page

```
[
  {
    id: bigint,
    title: string,
    price: double,
    rating: 5,
    vendor: {
      id: bigint,
      name: string
    }
  }
]
```

Get products by category

Usage: Landing page

```
[
  {
    id: category as string,
    products: [
      {
        id: bigint,
        title: string,
        price: double,
        rating: int,
        vendor: {
          id: bigint,
          name: string
        }
      }
    ]
  }
]
```

Get products by subcategory

Usage: Landing page

```
[
  {
    id: subcategory as string,
    products: [
      {
        id: bigint,
        title: string,
        price: double,
        rating: 5,
        vendor: {
          id: bigint,
          name: string
        }
      }
    ]
  }
]
```

Get product reviews

Usage: Product detail page

```
[
  {
    rating: double,
    text: string
  }
]
```

Get product detail by id

Usage: Product detail page

```
{
  id: bigint,
  title: string,
  description: string,
  piecesInStock: number,
  price: number,
  rating: number,
  vendor: {
    id: bigint,
    name: string
  },
  categories: [string],
  subcategories: [string]
}
```

Get orders for user

Usage: Customer profile

```
[
  {
    id: email as string,
    orders: [
      {
        id: bigint
        status: enum,
        price: number,
        created: timestamp,
        updated: timestamp,
      }
    ],
  }
]
```

Get order detail by id

Usage: Customer profile

```
{
  id: bigint,
  status: enum,
  price: number,
  address: {
    country: string,
    zipCode: string,
    city: string,
    street: string,
    houseNumber: string,
  },
  created: datetime,
  updated: datetime,
  products: [
    {
      id: bigint,
      title: string,
      description: string,
      price: number,
      vendor: {
        id: bigint,
        name: string
      }
    }
  ]
}
```

Get categories

Usage: Landing page

```
[
  {
    id: categoryname as string,
    description: string
    subcategories: [
      {
        name: string,
        description: string
      }
    ]
  }
]
```


3 Design

In this section, we summarize the information from the specification and formalize it into models based on which the system will be further implemented.

3.1 High-level Component diagram

This diagram models the distribution of services and their interconnection.

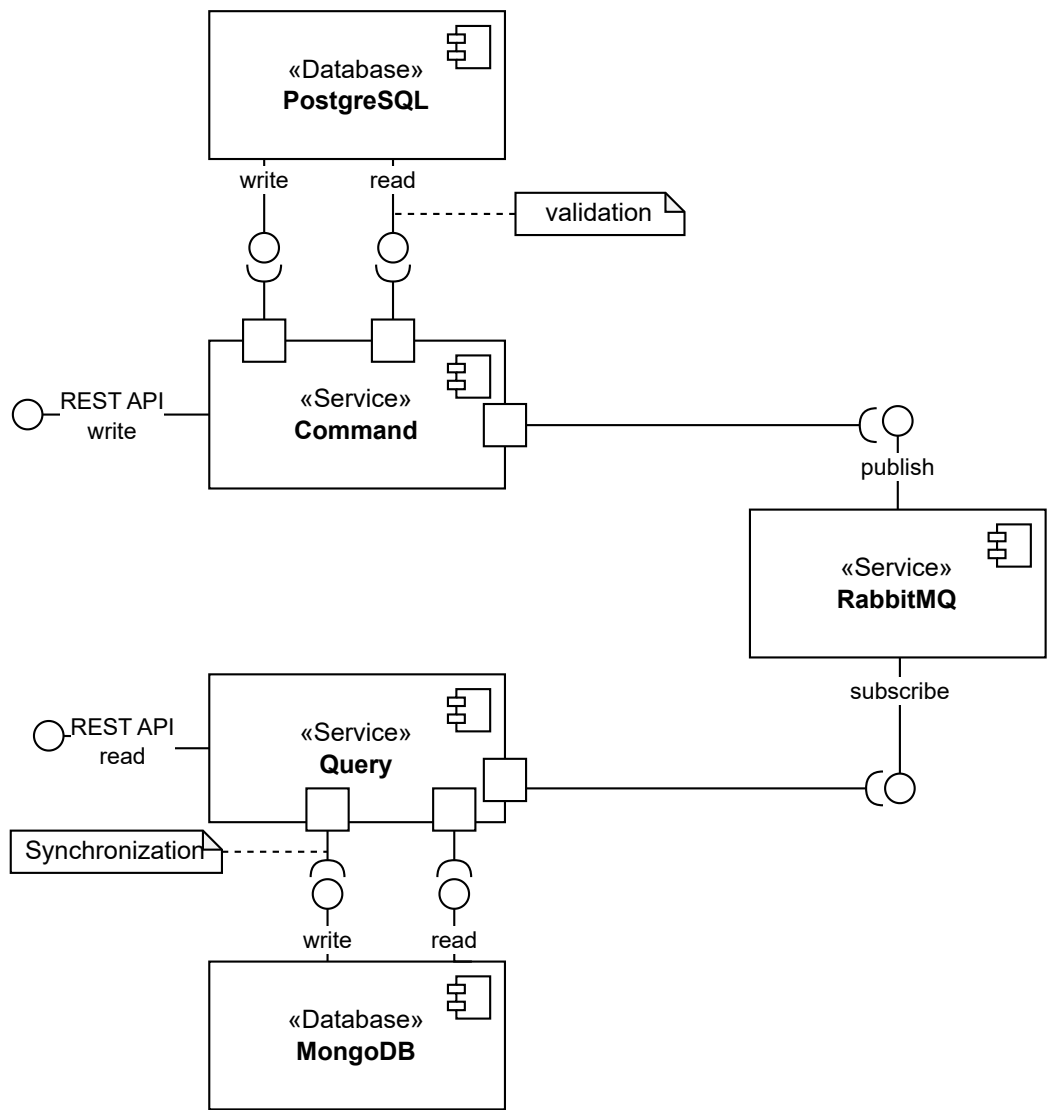


Figure 2: High-level Component diagram

3.2 Detail Component diagram

The detailed component diagram gives us a closer look at the system architecture. Where we are already dealing with the link between the individual packages and parts of the system at the code level.

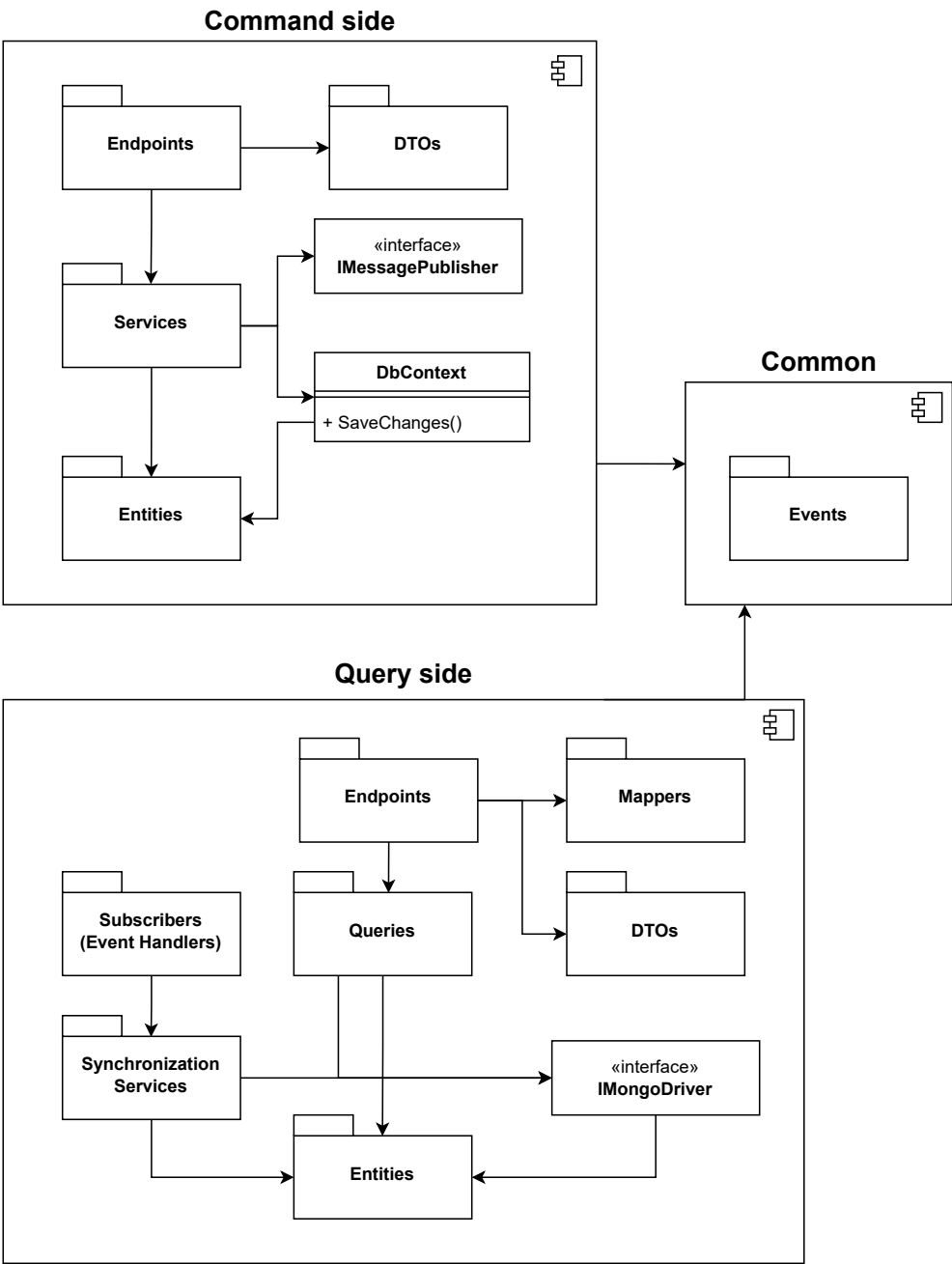


Figure 3: Detail Component diagram

3.3 Synchronising data stores

For synchronization between the relational and NoSQL database we chose the message broker Rabbit.MQ, within which we create a channel for each entity in the system. The message will contain the type of operation (create, edit, delete) and the necessary data to perform the NoSQL database update. Rabbit.MQ supports data persistence, so if the read service goes down, it will be delivered when it comes back up. Thus, on the relational database side, if the database is edited, the appropriate message is sent to the appropriate channel. Subsequently, on the read side, there will be a service for each channel that will provide the read service from the channel and edit any necessary collections so that the collections reflect the actual state in the relational database.

3.4 E-R Diagram

This diagram represents specific tables and data layouts within a relational database.

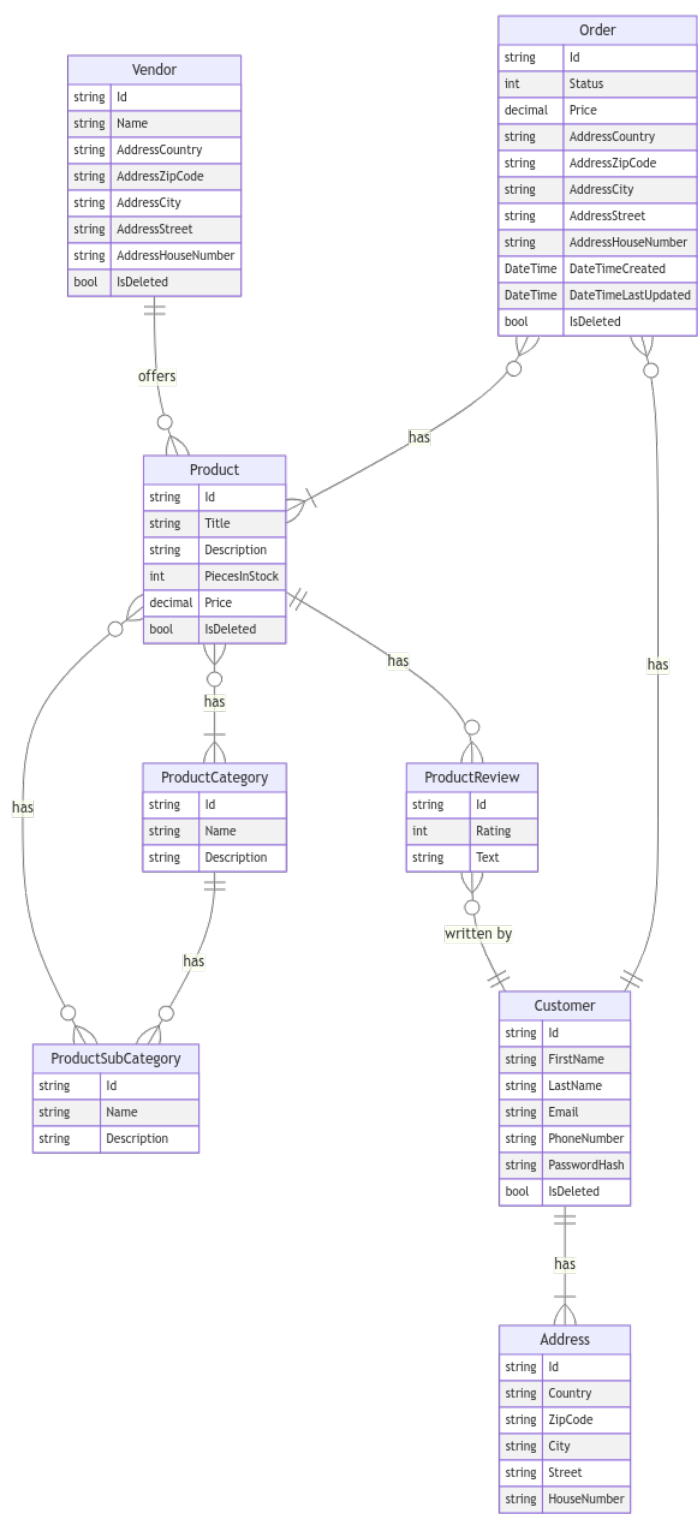


Figure 4: ER - Diagram

3.5 MongoDB

MongoDB is a document database system. It stores data in collections, where each collection consists of documents. It stores data in BSON (binary JSON) format. Collections are schema-less, which is useful for storing sparse data or for adding attributes dynamically. Proper database design is crucial for efficient searching, where we can use index support for more complex searches. MongoDB is also found in modern web stacks as the main persistent storage due to its support for transactional locking, which provides us with database consistency. MongoDB supports distributed deployments and the associated data replication and horizontal database scaling support.

3.5.1 Data validity period

The validity of data in the NoSQL database will directly correlate with the validity in the relational database, since the NoSQL database is primarily used here to streamline queries and reduce JOIN operations over relational data.

3.5.2 Scalability

Scalability is a key feature of MongoDB, without which we could not talk about it as a modern data store. It is a key feature for large and fast growing applications that require high availability. There are two main approaches to scaling the system. For large systems, it is advisable to combine these two approaches.

Horizontal scaling (Sharding)

With horizontal scaling, we spread the load across multiple nodes so that each node contains only a portion of the data. The distribution between nodes is done using a suitable sharding key. Then the query is sent to the router and it already knows the network tology and where to look for the answer to the query according to the provided sharding key.

Vertical scaling

Vertical scaling consists of increasing the hardware resources of a given node.

3.5.3 Definition of collections

In this section we will define the individual database collections and discuss how often the data will be accessed and why we chose the following layout for this application.

Collection of vendors

The collection contains information about vendors. We have embedded categories into the collection so it can be used when a customer views a vendor's profile. In the case of a large number of vendors, we can use a hashed shard key for distribution via the vendor name attribute.

```
[
  {
    _id: 1,
    name: "Vendor XYZ",
    category: ["Toys", "Pets"],
    address: {
      country: "The Czech Republic",
      zipCode: "561 64",
      city: "Jablone Nad Orlici",
      street: "Lesni",
      houseNumber": "280"
    }
  }
]
```

Collection of customers

Unlike the vendor table, we can expect a larger volume of data and faster data creation. We will use email as id, since this attribute must be unique. Since we won't need the address itself without the user we will get rid of one join and put the collection of addresses into the customer object. In case of a large number of customers, we can use a hashed shard key for distribution via the customer's email.

```
[
  {
    _id: "email@gmail.com",
    firstName: "Adam",
    lastName: "Johnson",
    phoneNumber: "00420123123123",
    addresses: [
      {
        _id: 1,
        country: "The Czech Republic",
        zipCode: "262 42",
        city: "Rozmital Pod Tremsinem",
        street: "U medvidku",
        houseNumber: "1563"
      }
    ]
  }
]
```

Collection of products

We expect this collection to contain a large amount of data. Since we expect significantly more reads than writes, we have embedded vendor, categories and subcategories into collection. There is also total rating calculated based on the reviews of this product. In the case of a large number of products, we can use a hashed shard key over the product name for distribution, allowing for quick searches by name. An index would be created on the vendor name to efficiently display the products of that vendor.

```
[
  {
    _id: 1,
    title: "Rat killer",
    description: "Poison for rats",
    piecesInStock: 1,
    price: 45.99,
    rating: 5,
    vendor: {
      _id: 1,
      name: "Extreminators",
    },
    categories: [
      {
        "name": "Pests"
      }
    ],
    subcategories: [
      {
        "name": "Poisons"
      }
    ]
  }
]
```

Collection of orders

The order collection will be used to display the details of a given order from the customer's profile. Furthermore, the collection will be used to display the list of orders of the given user. For this purpose, we will create an index on the customer id. In the case of a large number of orders, we can use a hashed shard key for distribution through the customer, which will guarantee that all orders of one customer will be on just one node.

```
[
  {
    _id: 1,
    customerId: 1,
    status: "paid",
    price: 49.90,
    address: {
      country: "Czech Republic",
      zipCode: "612 00",
      city: "Brno",
      street: "Bozetechova",
      houseNumber: "2/1"
    },
    created: "2023-10-16T15:30:00Z",
    updated: "2023-10-17T15:30:00Z",
    isDeleted: "false"
    products: [
      {
        _id: 1,
        title: "Rat killer",
        description: "Poison for rats",
        price: 45.99,
        vendor: {
          _id: 1,
          name: "Extreminators",
        },
      },
    ],
  },
]
```

Collection of reviews

The review collection will contain an index to the product id, as we will primarily want to calculate reviews of specified product. In the case of a large number of reviews, we can use a hashed shard key to distribute them across the product, guaranteeing that all reviews of a single product will be on a single node.

```
[
  {
    _id: 1,
    productId: 1,
    rating: 5,
    text: "Very good product"
  }
]
```

Collection of products of category

Collection of products grouped by category for quick loading of products in the product catalog on the client side. This collection was designed purely for performance optimization of reading products (we do not need to apply filtering). Vertical scaling is not anticipated to be required for this collection.

```
[
  {
    _id: "CategoryName",
    products: [
      {
        _id: 1,
        title: "Rat killer",
        description: "Poison for rats",
        price: 45.99,
        rating: 4,
        vendor: {
          _id: 1,
          name: "Extreminators",
        },
      },
    ]
  }
]
```

Collection of products of subcategory Collection of products grouped by subcategory of given category for quick loading of products in the product catalog on the client side. This collection was designed purely for performance optimization of reading products (we do not need to apply filtering). Vertical scaling is not anticipated to be required for this collection.

```
[
  {
    _id: ("ParentCategory", "SubCategory"),
    products: [
      {
        _id: 1,
        title: "Rat killer",
        description: "Poison for rats",
        price: 45.99,
        rating: 4,
        vendor: {
          _id: 1,
          name: "Extreminators",
        },
      },
    ]
  }
]
```

Collection of categories

This collection will be used on the landing page to display available categories of the products. We will insert the subcategory information directly into the collection record. We expect this collection to be stable (basically without modifications). Vertical scaling is not anticipated to be required for this collection.

```
[
  {
    _id: 1,
    categoryName: "name",
    description: "description of category",
    subCategories: [
      {
        _id: 2,
        categoryName: "name"
      }
    ]
  }
]
```

4 Technologies

The system will be implemented using ASP.NET Core framework. Both read and write services will be implemented as web API. Communication between services will be handled by message broker RabbitMQ. Individual services will run in docker containers and we will use docker compose for deployment. Postgres database has been chosen as the relational database and MongoDB will be used on the read side of the service due to its versatility.

5 Deployment

The application is ready to be deployed within the Docker runtime environment, which enables easy deployment without the need to deal with dependencies and ensures isolated running of individual applications. Each application, service, and database is deployed within its own container where the docker compose tool is used to orchestrate the containers. The structure of each container is shown in the deployment diagram. Thus, to start the system, it is only necessary to have the docker daemon installed and running, and to start the application using the `docker compose up` command from the root directory, which contains the `docker-compose.yml` file. The Write Service is available at `http://127.0.0.1:5000`, `https://127.0.0.1:5001` and the Read Service at `http://127.0.0.1:5002`, `https://127.0.0.1:5003`. Both applications have an interactive OpenApi specification, which is available at the `/swagger` endpoint. From the root Write Service folder you need to run the database migration using `dotnet ef database update`. For this you will need the dotnet command line tool and the Entity Framework Design package.

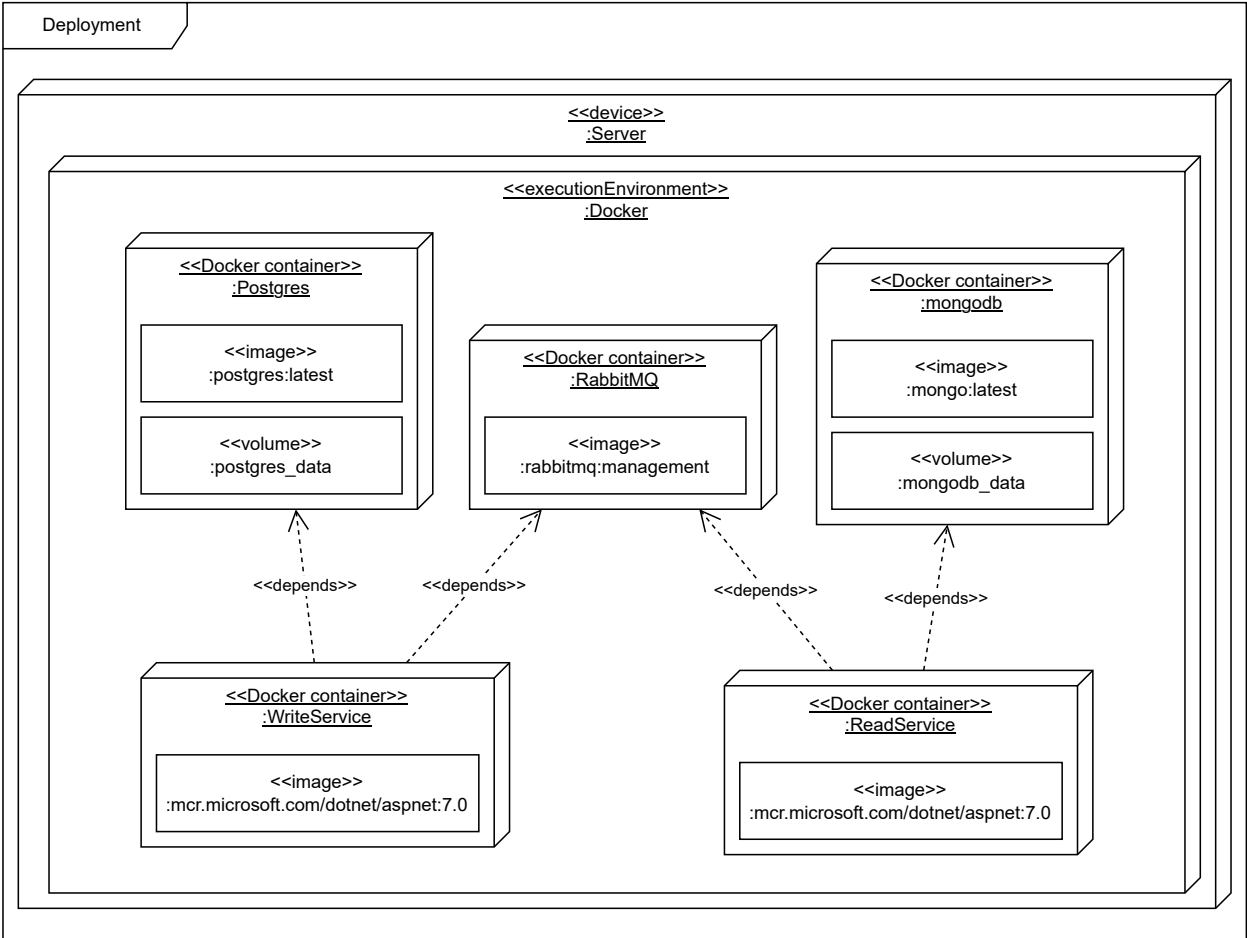


Figure 5: Deployment Diagram

6 Testing

As part of the testing, automatic end-to-end tests were implemented using the `xunit` library. Applications are run inmemory using `WebApplicationFactory` and individual endpoints are called over them. Separate databases are used for testing. Each test is a scenario that involves creating or modifying data via the Write service, and then testing that the data has indeed been written to the Read service. It also tests whether the application adheres to consistency rules, such as making it impossible to order items that are already deleted or out of stock.

Test	Duration
▲ ✓ CQRS.EndToEndTests (8)	27.2 sec
▲ ✓ CQRS.EndToEndTests.Tests (8)	27.2 sec
▲ ✓ CustomerTests (5)	17.8 sec
✓ TestAddAndReadCustomerWithAddress	3.5 sec
✓ TestDeleteCustomer	3.4 sec
✓ TestDeleteCustomerAddress	3.6 sec
✓ TestUpdateAndReadCustomer	3.5 sec
✓ TestUpdateCustomerAddress	3.7 sec
▲ ✓ OrderTests (3)	9.5 sec
✓ CompleteOrder	4.3 sec
✓ CreateDuplicityNewOrder	3.4 sec
✓ TestCompleteOrderWithOutOfStockPr...	1.8 sec

Figure 6: Result of automatic tests

The rest of the use cases were tested using the **Postman** application, which allows us to import the definition of individual services in the **OpenApi** format, which is made possible by the **Swagger** library. The **Write Service** call was manually tested and then whether the change was correctly propagated to the **Read Service**. Then it was manually verified using **pgAdmin** that the data was correctly modified within the relational database.


<div>  Swagger <small>powered by SMARTBEAR</small> </div> <div> Select a definition WriteService v1 </div>	
<div> WriteService 1.0 OAS3 <small>http://localhost:5009/swagger/v1/swagger.json</small> </div>	
CategoryEndpoints	
POST	/api/categories
PUT	/api/categories/{categoryId}
DELETE	/api/categories/{categoryId}
CustomerEndpoints	
POST	/api/customers
PUT	/api/customers/{customerId}
DELETE	/api/customers/{customerId}
POST	/api/customers/{customerId}/addresses
PUT	/api/customers/{customerId}/addresses/{addressId}
DELETE	/api/customers/{customerId}/addresses/{addressId}
OrderEndpoints	
POST	/api/orders
POST	/api/orders/{orderId}/add-to-cart/{productId}

Figure 7: OpenAPI Specification Write Service

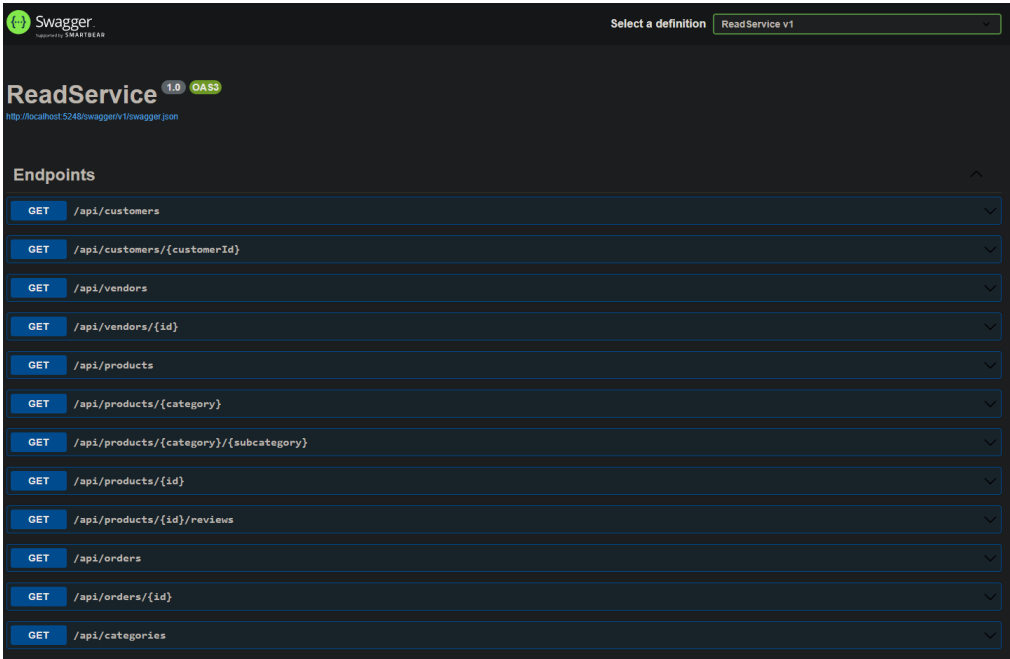


Figure 8: OpenAPI Specification Read Service

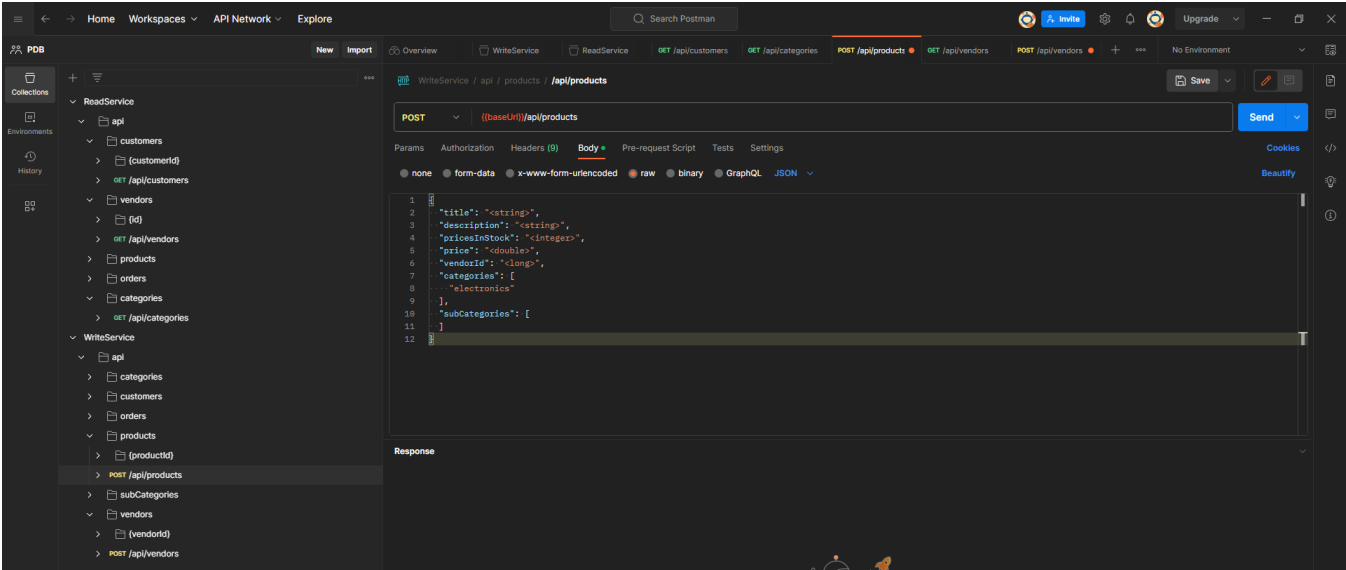


Figure 9: Postman workspace

7 Conclusion

The goal of this project was to test the principles associated with the implementation of the CQRS patter, so we did not consider user authentication and the absence of other functionalities that would certainly be needed to run a similar system in a real environment. However, the system as such works despite its limitations and both databases synchronize as expected.