Bachelor Thesis

Evaluating Data Management Patterns in Microservice-Based Applications

Author

Eylül Gökce Harputluoglu

desired academic degree

Bachelor of Science (BSc)

Vienna, 01.02.2021

Student ID:	A11728483
Subject Area:	Computer Science - Medical Informatics
Supervisors:	Univ.-Prof. Dr. Uwe Zdun
	BSc MSc Evangelos Ntentos

Acknowledgements

Evangelos Ntentos gave me valuable guidance through the project. I would like to thank Mr. Ntentos for providing useful reading materials and for supporting the design of all the patterns by giving beneficial feedback. I also would like to thank Professor Dr. Uwe Zdun for giving me an opportunity to do this project in the Software Architecture Research Group. I would like to thank to my sister Hande Harputluoglu as well for the motivational support during this project.

Contents

1 Microservices **4**
- 1.1 Monolithic vs. Microservices Architecture 4
- 1.2 Benefits of Microservices Architecture 4
- 1.3 Goal of the Project 5

2 Related Work **6**
- 2.1 Example of Microservices Based Application: Amazon 6

3 Patterns **7**
- 3.1 Database per Service 7
- 3.2 Shared Database 7
- 3.3 Event-Based Communication: Message Broker Pattern 8

4 Case Study **10**
- 4.1 Technology Stack 10
- 4.2 Description: Web Shop 10
  - 4.2.1 Customer Service 10
  - 4.2.2 Product Service 10
  - 4.2.3 Cart Service 10
  - 4.2.4 Order Service 11
  - 4.2.5 Payment Service 11
  - 4.2.6 Notification Service 11
- 4.3 Prototypes 11
  - 4.3.1 Prototype 1: Database per Service 12
  - 4.3.2 Prototype 2: Shared Database 16
  - 4.3.3 Prototype 3: Message Broker 20

5 Evaluation **24**
- 5.1 Test Cases 24
- 5.2 Results 25
- 5.3 Discussion of the results 29

6 Conclusion **30**

7 Future Work **30**

1 Microservices

Microservices is an architectural and organizational approach to software development where an application consists of small independent or loosely-coupled services that communicate over well-defined APIs.[19] They can be implemented by using different programming languages, databases, hardware and software environment, depending on what fits best for its purpose.[5] These services are owned by small teams. They are maintainable and testable. Microservices architectures make applications easier to scale and faster to develop.[5] Microservices are used widely by the software companies such as Amazon, Netflix, Ebay, Uber and more.

1.1 Monolithic vs. Microservices Architecture

With monolithic architectures, processes are not independent and they run as a single service.[19] If there is a problem or a need about one process, entire architecture may be affected or changed. With time, the code becomes increasingly complex, and adding new features or updating the existing ones starts costing too much time and effort.[7]

In microservices, adding or improving features is easier. They are built for business capabilities and each service performs a single function. Services are independent from each other and they can be updated, changed, or scaled depending on the needs of the service.[19, 4]

1.2 Benefits of Microservices Architecture

Microservices have the following features which are important and beneficial in software development[10]:

- With microservices architectures, application is easier to understand, develop and test.
- Each service is owned by a small team. As services are function-specific and independent of each other, teams can focus on only one service and work more quickly, thus the development cycle time gets shorter.
- Microservices allow each service to be independently scale and maintain availability when a service experiences a high demand.
- Microservices facilitate continues integration, continuous delivery and deployment [1]. It makes it easier to try out new ideas and reverse them if they don't work as expected.
- For each service, teams are free to choose best tools to solve their specific problems without following a strict procedure or design patterns. Each service can use its own language, framework etc.

- Each service is responsible for one function, thus it enables teams to use functions for multiple purposes.
- In microservices, application's failure is less probable. In a monolithic architecture, if a single component fails, it can cause the entire application to fail which is not the case with microservices. Additionally, as microservices make it easier to debug and test the applications, the probability to have an error-free application is improved. [3]

1.3 Goal of the Project

The goal of this project is to implement a web shop application based on microservices. Three data management patterns will be used in implementation and three prototypes will be created by using the corresponding patterns. The prototypes will be tested in different scenarios and their performances will be compared. The objective of this project is to understand data management patterns and their use in microservices design. At the end of this project, it is expected to see the impact of structural patterns on the performance of the application.

2 Related Work

Today many service-based systems follow the microservice architecture model. Microservice architecture is usually used to build distributed systems and to have independent service development and loosely coupled dependencies between services. There are many microservice and data management patterns and depending on the system properties, the most suitable one can be choosen.[6]

Besides the patterns used in this project, there are more patterns which can be implemented for the microservice architecture, such as decomposition patterns, additional data management patterns, messaging and remote procedure invocation patterns, deployment patterns, different patterns for communication style, UI patterns, observability patterns, access token, circuit breaker, testing patterns, cross-cutting concerns patterns and many more. [17]

2.1 Example of Microservices Based Application: Amazon

Amazon is a technology company which focuses on e-commerce, cloud computing, digital streaming and artificial intelligence. It started as an online marketplace for books but expanded to sell electronics, software, video games, etc.[1]

Amazon is one the first companies which transformed its system into microservices. Before, Amazon's architecture was considered as a monolith since all the components in the system are tightly coupled. As most of the companies, Amazon took a monolith-first approach in the beginning since it was custom and very quick way to start. However, over the time as the company got bigger, there were more developers working on the project, the code base got larger and architecture got complex.[8] To be able to change a feature, the whole system needed to be rebuilt, the test cases needed to be rerun. Major code changes were stuck in the deployment pipeline for weeks before the customers could use them.[9] Therefore, there was a need to separate the services so that software development life cycle gets faster. To solve these problems, they had a principle which each function should have a single purpose and they could only communicate through their own APIs. This is also the main principle behind microservices architecture. Breaking down structures into single applications allowed developers to see where the bottlenecks are, therefore shorten the pipeline. Moving to the Amazon Web Services (AWS) cloud allowed Amazon to scale up or down when necessary, reduce the number and duration of outages, and save money.[12]

3 Patterns

In order to evaluate and compare the performance of the microservice-based application, there are three different data management patterns which are implemented in this bachelor project. These patterns are database per service, shared database for all services and an event-based communication between services.

3.1 Database per Service

The database per service model supports the idea that it should keep each microservice's persistent data private to that service and this data should be accessible only via its API. A service's database is part of the service implementation; therefore it cannot be accessed by other services. In this model, a service's transactions involve only its database. [15]

Using a database per service pattern has several benefits:

- The services are loosely coupled which means that each of its components has little or no knowledge about other components. If there is a problem or a change in one of the databases, the other services won't be affected by that.
- Each service can use a specific type of database which is appropriate to its needs and functions.

Using a database per service has also a few drawbacks[15]:

- Some data can exist in multiple databases and since these databases are independent and can be in form of different types, it is challenging to implement queries which use these shared data.
- Managing multiple SQL and NoSQL databases is complex.
- Implementing business transactions that extend to multiple services is not straightforward.

While using this pattern in this project, problems due to these drawbacks were faced in the implementation, especially where it was necessary to use a join operation between different service databases.

3.2 Shared Database

The second database architecture used in the implementation is shared database. The idea behind this architectural design is that each service freely accesses data owned by the other services by using local ACID (atomicity, consistency, isolation, durability) transactions[16]. All services are using the same database in the system.

This pattern has following benefits in the microservice-based applications[16]:

- A developer uses familiar and straightforward ACID transactions to ensure data consistency.
- A single database is simpler to work on.

A shared database for the microservices has also few drawbacks[16, 18]:

- There is a requirement for development time coupling to ensure data consistency: A developer works on one service and this service needs to coordinate schema changes with the developers of other services that access the same tables. This coupling and additional coordination will slow down development.
- There is a requirement for runtime coupling. Since all services access the same database, they can potentially interfere with one another. For example, if long running service transaction holds a lock on one table in the database, then the other services which need to access this table will be blocked until long running service transaction releases the lock. Otherwise, there will be inconsistency between table content which can cause problems.
- Single database might not satisfy the data storage and access requirements to this database from different services may be different.
- In shared databases, it is harder to extend the application with new services.

In order to support ACID principle and to avoid possible problems, database transaction methods was implemented inside of the Order Service to make an order disposable.

3.3 Event-Based Communication: Message Broker Pattern

The previous patterns were based on synchronous communication with using REST API (REpresentational State Transfer) between services which means the client forwards a REST request and actively waits for a response. Generally, in synchronous communication, the client prepares the message and transmits it over the network to the server. It waits for a response and while waiting it does not perform any other processing as part of its computing thread. [11]

As the third pattern, an event-based communication between microservices has been chosen. In order to realize an event-based communication, message broker pattern was used in the implementation of this project. This model connects microservices through the transmission of messages while using a broker. A microservice publishes an event when something notable happens, such as when

it updates a business entity. Other microservices subscribe to those events and get notified when there is a change in these events. When a microservice receives an event, it can update its own business entities accordingly. This update may lead to more events being published. This is the essence of the eventual consistency concept. This publish/subscribe system is usually performed by using an event bus. The event bus can be designed as an interface with the API to subscribe and unsubscribe to the events publish events.[14, 13, 2]

Thanks to this pattern, the asynchronous communication can be built between services that the client can send a message/request, but he does not have to wait for a response.

This model has following benefits [2, 11]:

- The components may have different models of communication. With the Message Broker, communication between the components is supported and possible. Message queue, request-response mode, broadcast mode are examples of good use of Message Broker with different models of communications.
- Message Broker Pattern supports different message formats like JSON, Protobuf etc.
- The broker communication approach is ideal for one-way calls: sending message and not waiting for the response. It helps to balance the load on individual layers of the system and thus supports peak demands in an inbound traffic.
- There is no risk of overloading the message consumers.
- This pattern also allows you to save messages permanently for a better reliability, or only process them in memory for better efficiency.

Using a message broker has also several disadvantages[11]:

- Modelling the application using messages is more complicated.
- One of the message broker technologies should be set up in the system. However as broker is an additional object in the architecture, it is another potential failure point in the system. With broker use, there may be problems related to quorum behaviour, delivery guarantees, idempotency verification etc.[11]
- There is a possibility of overloading the queues or message broker when the message consumer is not available for a while.
- Message broker introduces delays in communication because broker runs in the memory mode and execution can take some time if the model exists in the disk.

In this project, RabbitMQ is used in order to build the event-based communication between services.

4 Case Study

In this project, there is one microservice-based application idea, a web shop application based on microservices is implemented by using three different patterns.

4.1 Technology Stack

All three patterns are implemented with using C# programming language. Therefore, the .Net Core framework is used for the microservice implementation. For the API gateway, Ocelot is used for the .Net Core application. The databases of all microservices are created in MySQL Database. For the third pattern (message broker pattern), an additional technology which is RabbitMQ is used. For the testing the applications, Apache JMeter is used.

4.2 Description: Web Shop

The main application idea of this project is building a web shop, where the customer of the online shop is the client. A web shop requires a customer service, a product service, a cart service, an order service, a payment service and a notification service. These services are implemented with microservices. The client can easily register to the system, select a product, add it to the cart and order the cart. At the end he virtually pays the total amount of the products from his cart. The customer receives notifications about errors as well as information messages.

4.2.1 Customer Service

The customer service is designed for handling the customer information by only connecting to the customer database. In the customer service, either a new customer is registered to the system, or customer id is found by using customer information.

4.2.2 Product Service

The product service is designed for handling all the products and their quantities in the storage. Inside of this service, it is possible to list all the products and their information, select a product and if the product is sold then update the quantity in the database.

4.2.3 Cart Service

The cart service does not use an actual database nevertheless it saves customer IDs and all carts belonging to the customers. Customers can insert new products to their cart, and when they go to their cart, all the selected products are listed.

4.2.4 Order Service

The order service is the most complicated service in the whole application. The client can order the entire cart where different products with different quantities are listed. Once a new order is created, the service gets the all products and calculates the total price. To be able to calculate the total price, order service finds products' prices by using their product ID and it simply multiples products' prices with desired number of units and adds them. It updates the product database with the quantities of product which will change after the purchase and finally saves all products and the total price with this order ID. To avoid any possible problems while doing all these processes, the database transaction method is used in the implementation. The client can get all of its orders or get all products by giving an order ID.

4.2.5 Payment Service

The payment service is responsible of the payment of the total amount for the order. We can insert a payment inside of this service and after the order is paid, the payment is saved to the database. It is also possible to list all previous payments inside of the payment service.

4.2.6 Notification Service

Notification service has three principle functions: adding a new notification, dismissing the current notification and listing all the notifications. All other services are using notification services to send a notification to the client. In this system, it is possible to get errors. These errors can appear because of an inconsistent input or data which will throw exceptions in the implementation or they can be HTTP-REST errors. Notification service gets errors, which are originated in this service or in the other services. It translates errors in a human-readable manner so that clients can read and understand the notifications.

For example, a customer tries to buy a product with 100 units but there are not enough units in the storage. Therefore, an error appears in product service which is later directed to notification service. Notification service gives a notification to the client about this insufficient units for the desired product. Notification service does not only send notification about the errors but also about acknowledgement and updates. If the customer orders the products and executes the payment without any problem, then the service notifies the customer with an information message about the success of the order.

4.3 Prototypes

There are three different patterns described in the previous chapter 3 and in this project, all these patterns are implemented as a separate prototype.

4.3.1 Prototype 1: Database per Service

The first prototype has database per service pattern implementation. Customer service, product service, order service and payment service have their own MySQL databases. For any operations, the services need to access to their databases. In the following figures, a class diagram, a component diagram and a sequence diagram are presented. The original images can be found here:

- Class Diagram for Database per Service
- Component Diagram for Database per Service
- Sequence Diagram for Database per Service

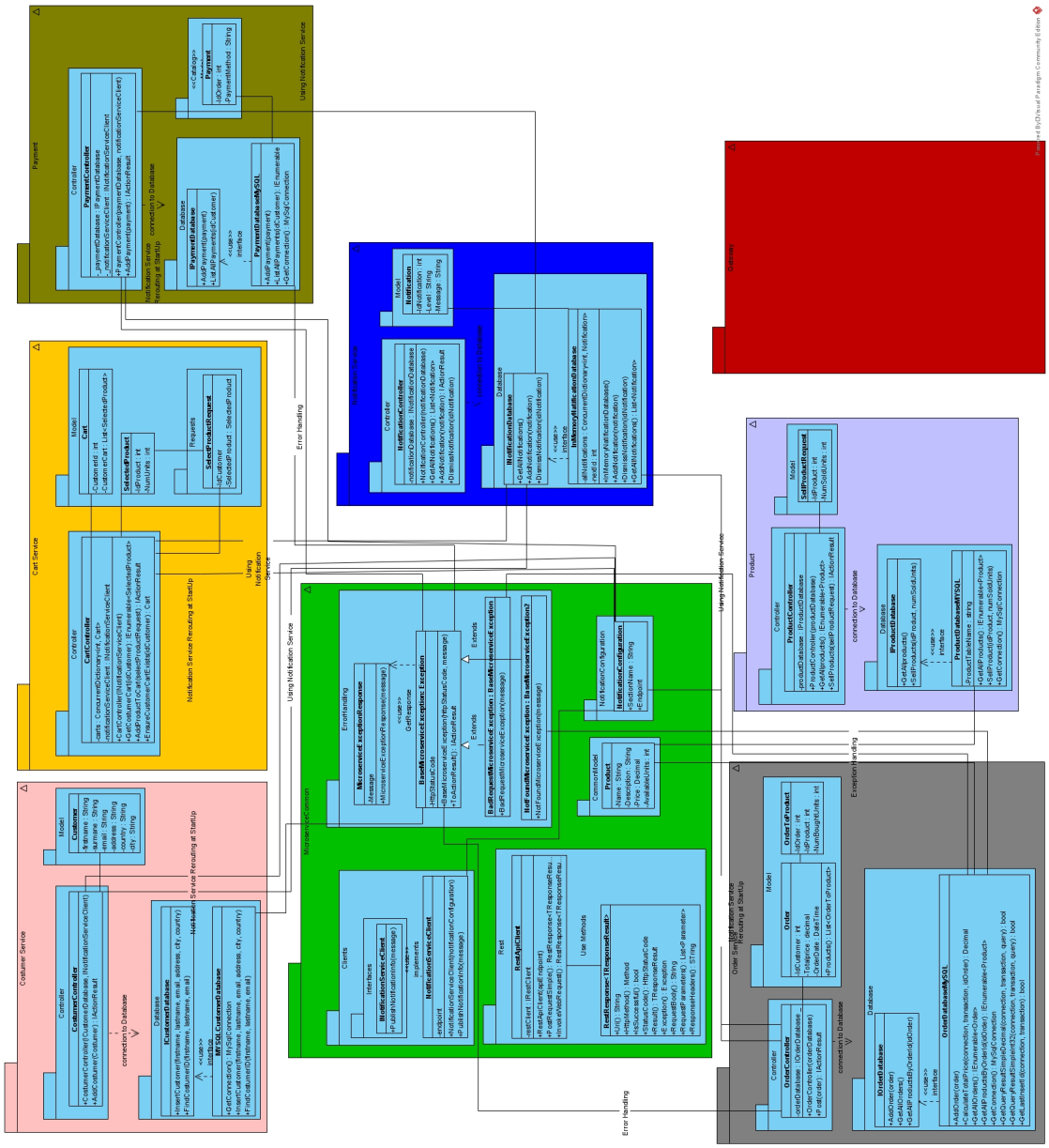


Figure 1: Class Diagram for Database per Service

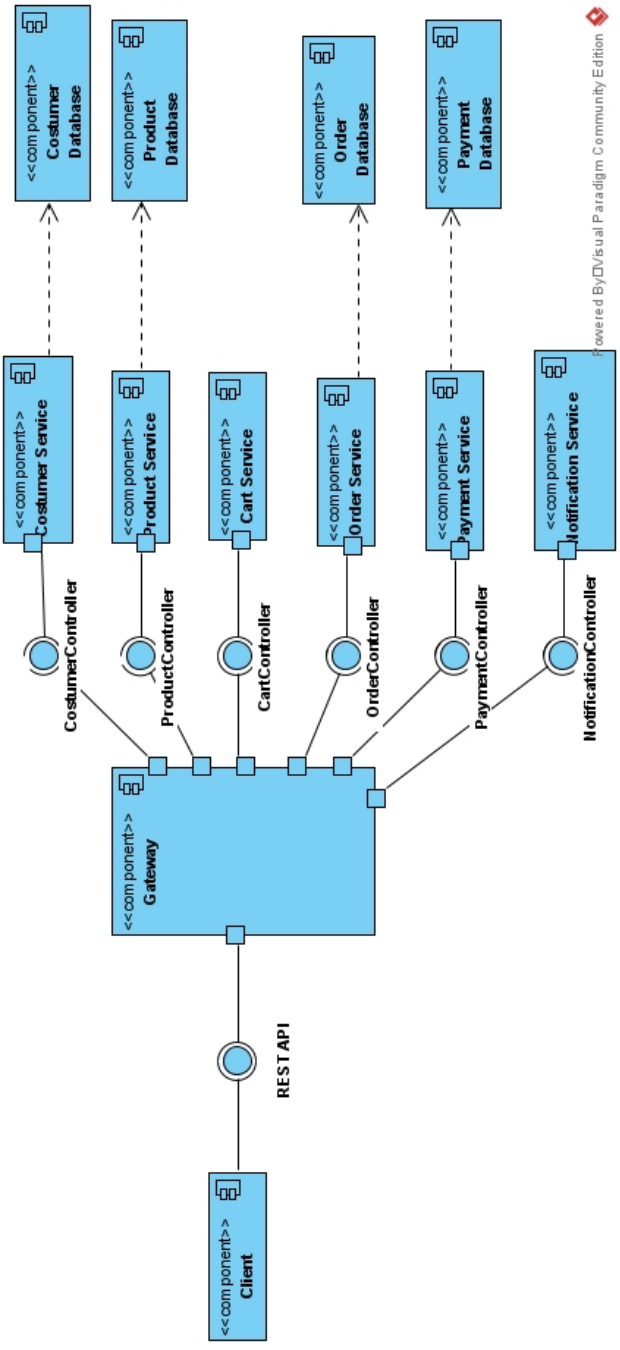


Figure 2: Component Diagram for Database per Service

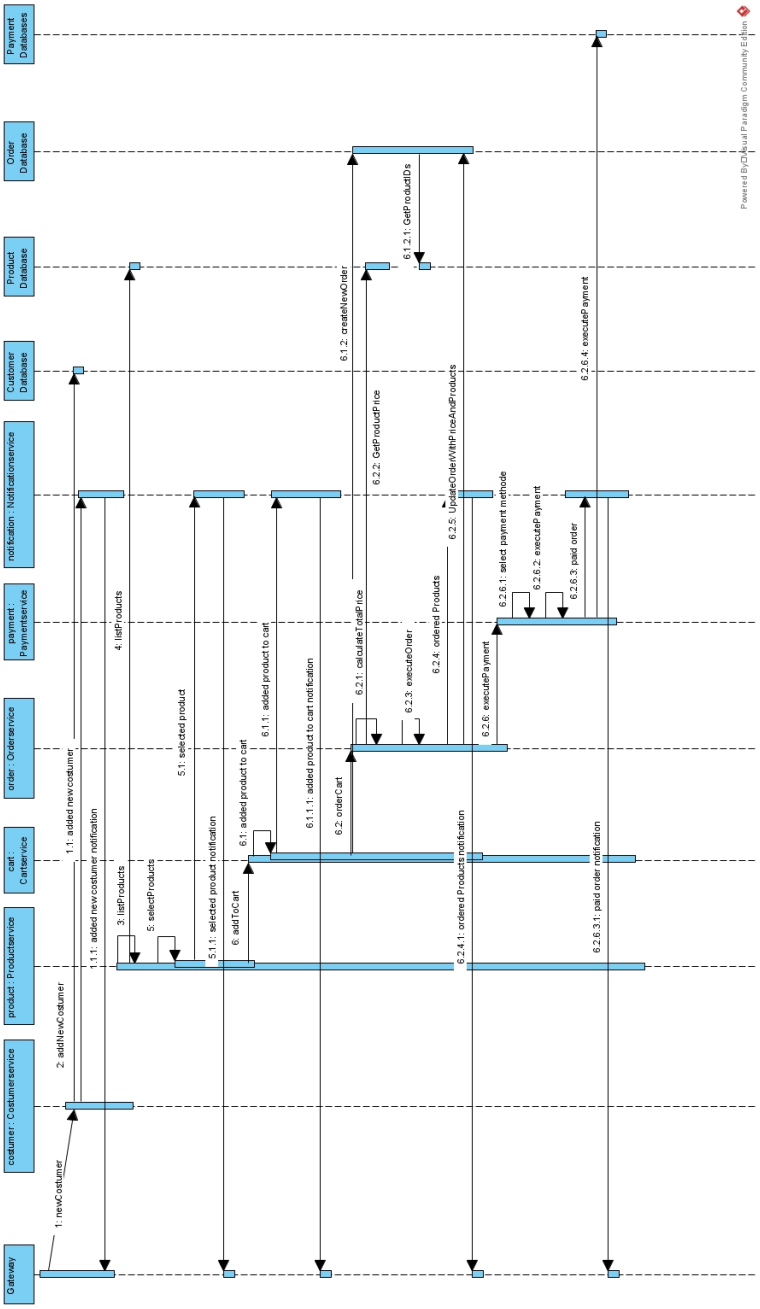


Figure 3: Sequence Diagram for Database per Service

4.3.2 Prototype 2: Shared Database

The second prototype has a shared database implementation. Customer service, product service, order service and payment service use the same MySQL database. For any operations, the services need to access to a single database. The rest of the system is same as the first prototype. In the following figures, a class diagram, a component diagram and a sequence diagram are presented. The original images can be found here:

- Class Diagram for Shared Database
- Component Diagram for Shared Database
- Sequence Diagram for Shared Database

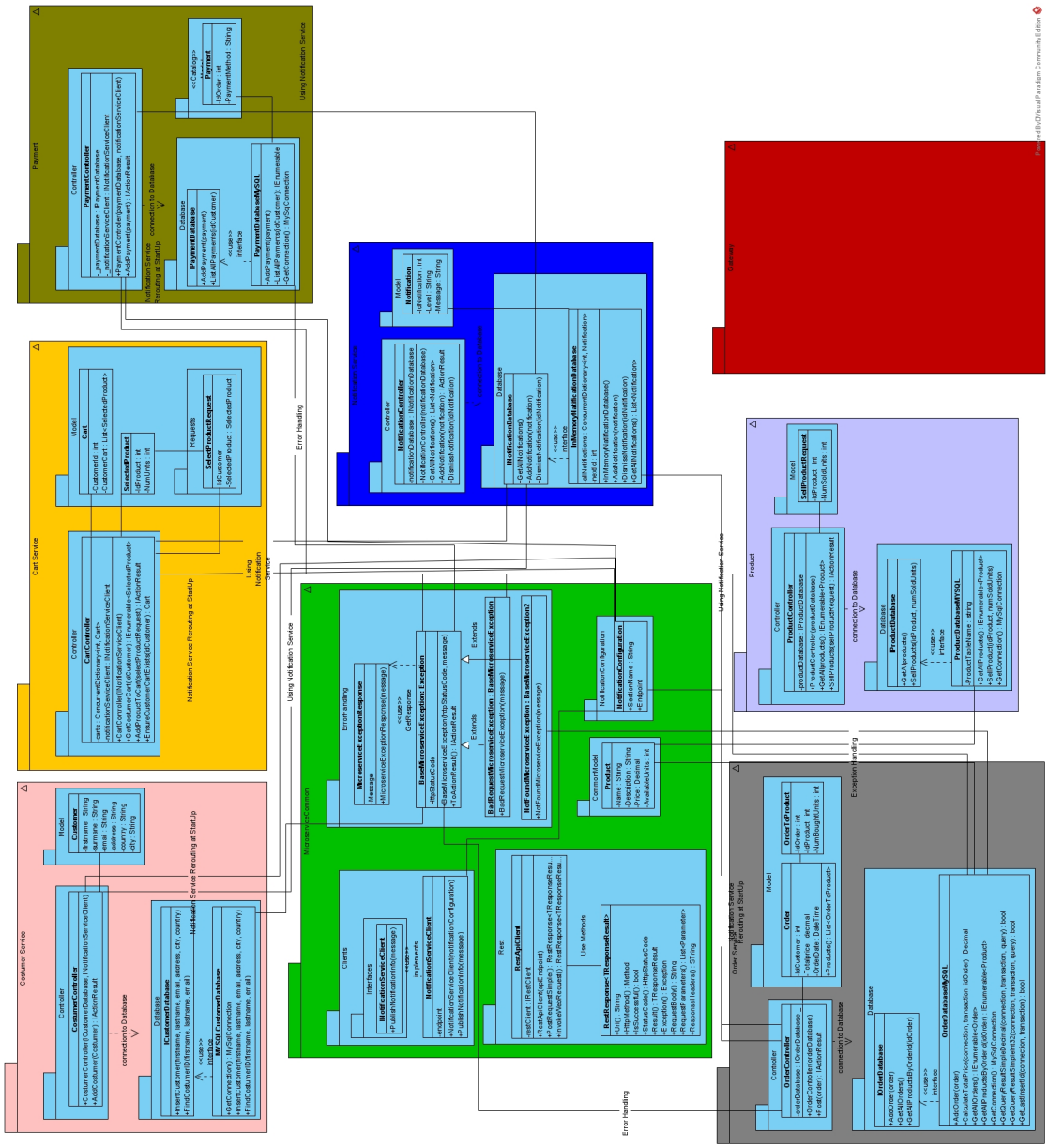


Figure 4: Class Diagram for Shared Database

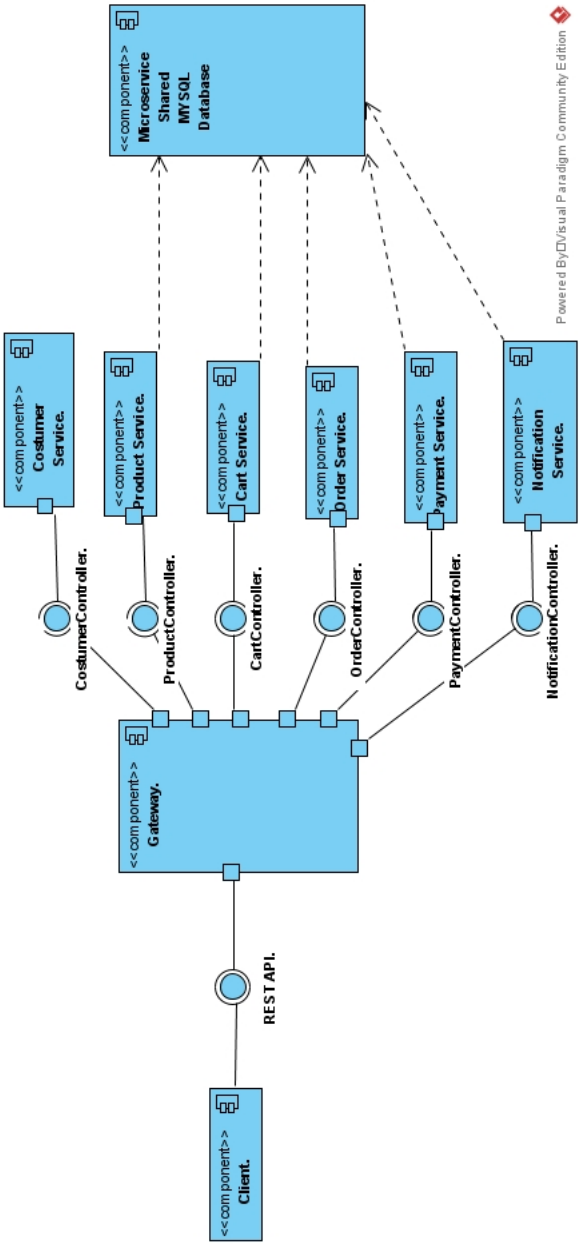


Figure 5: Component Diagram for Shared Database

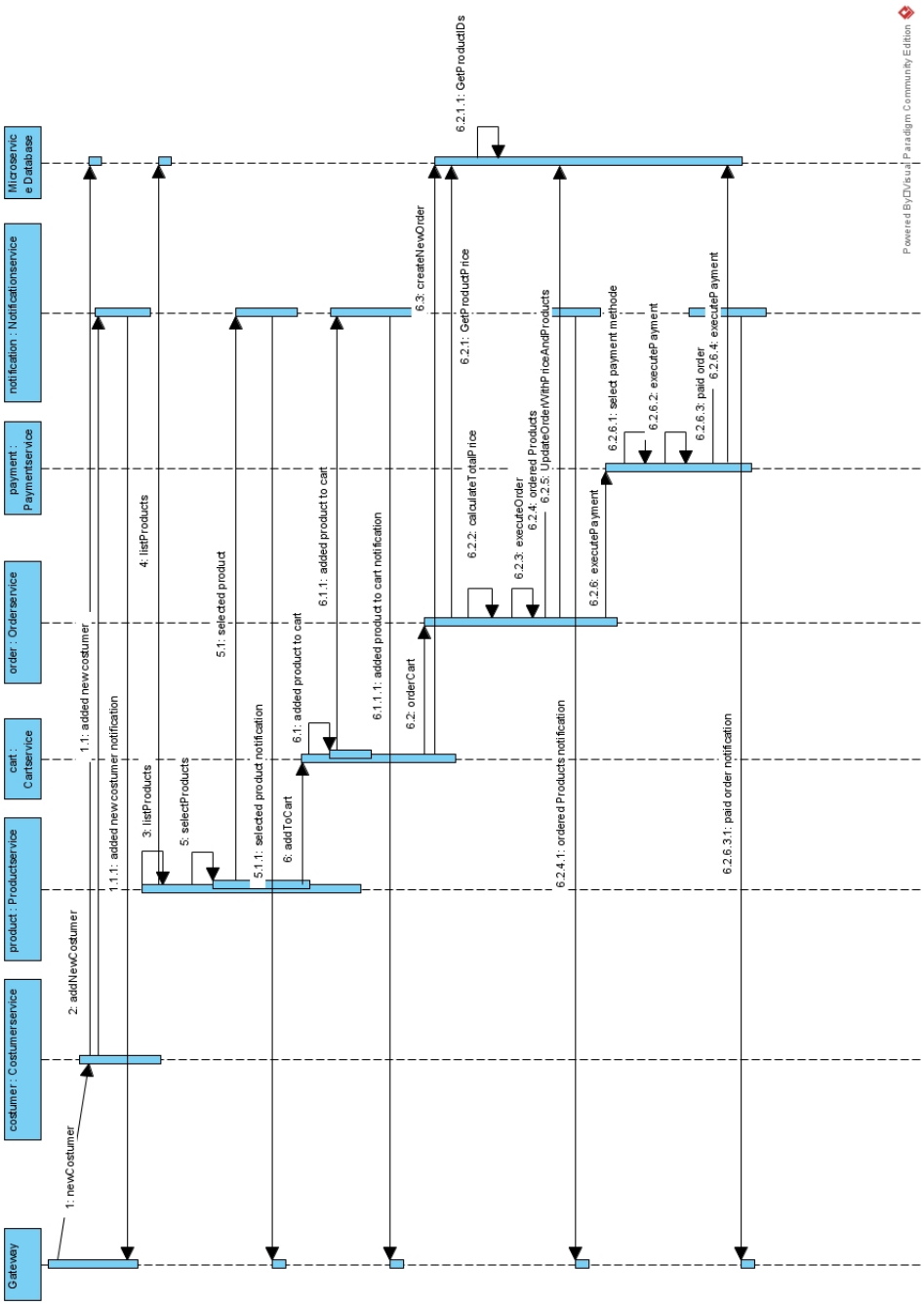


Figure 6: Sequence Diagram for Shared Database

4.3.3 Prototype 3: Message Broker

The third prototype has a event-based communication implementation. The message broker pattern has been implemented in this prototype. Cart service, order service and payment service are part of the communication within message bus. The cart service inserts selected products to the cart and publish an event. The order service will be activated if the service receives an order. For each order, a payment should be executing. Once the payment service receives a payment, the service will pay the amount of the order. In the following figures, a class diagram, a component diagram and a sequence diagram are presented. The original images can be found here:

- Class Diagram for Message Broker Patter
- Component Diagram for Message Broker Pattern
- Sequence Diagram for Message Broker Pattern

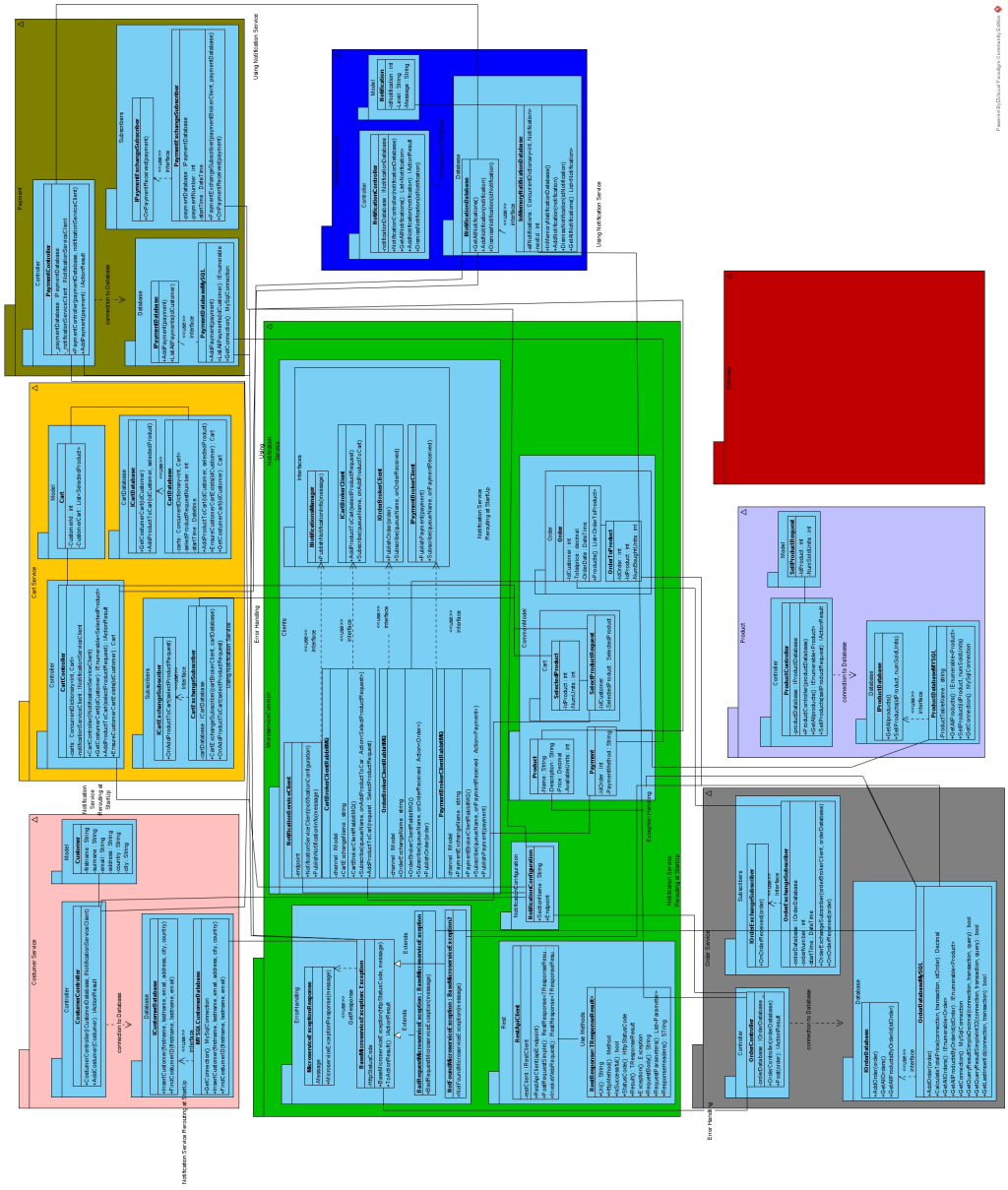


Figure 7: Class Diagram for Message Broker Pattern

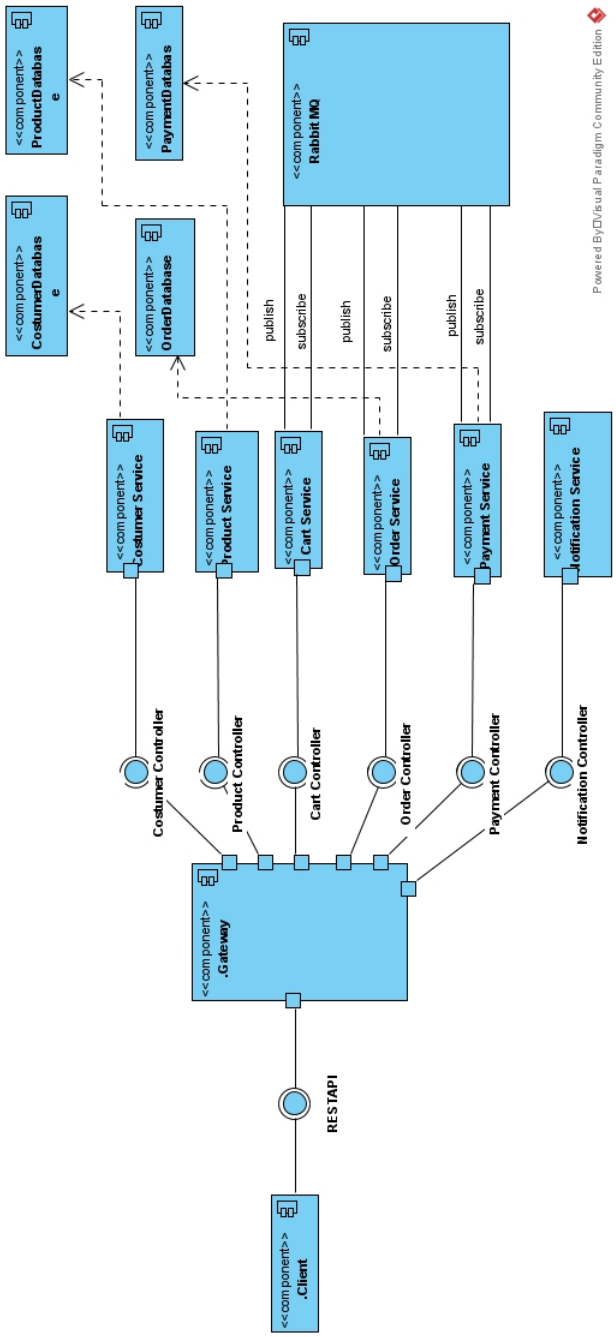


Figure 8: Component Diagram for Message Broker Pattern

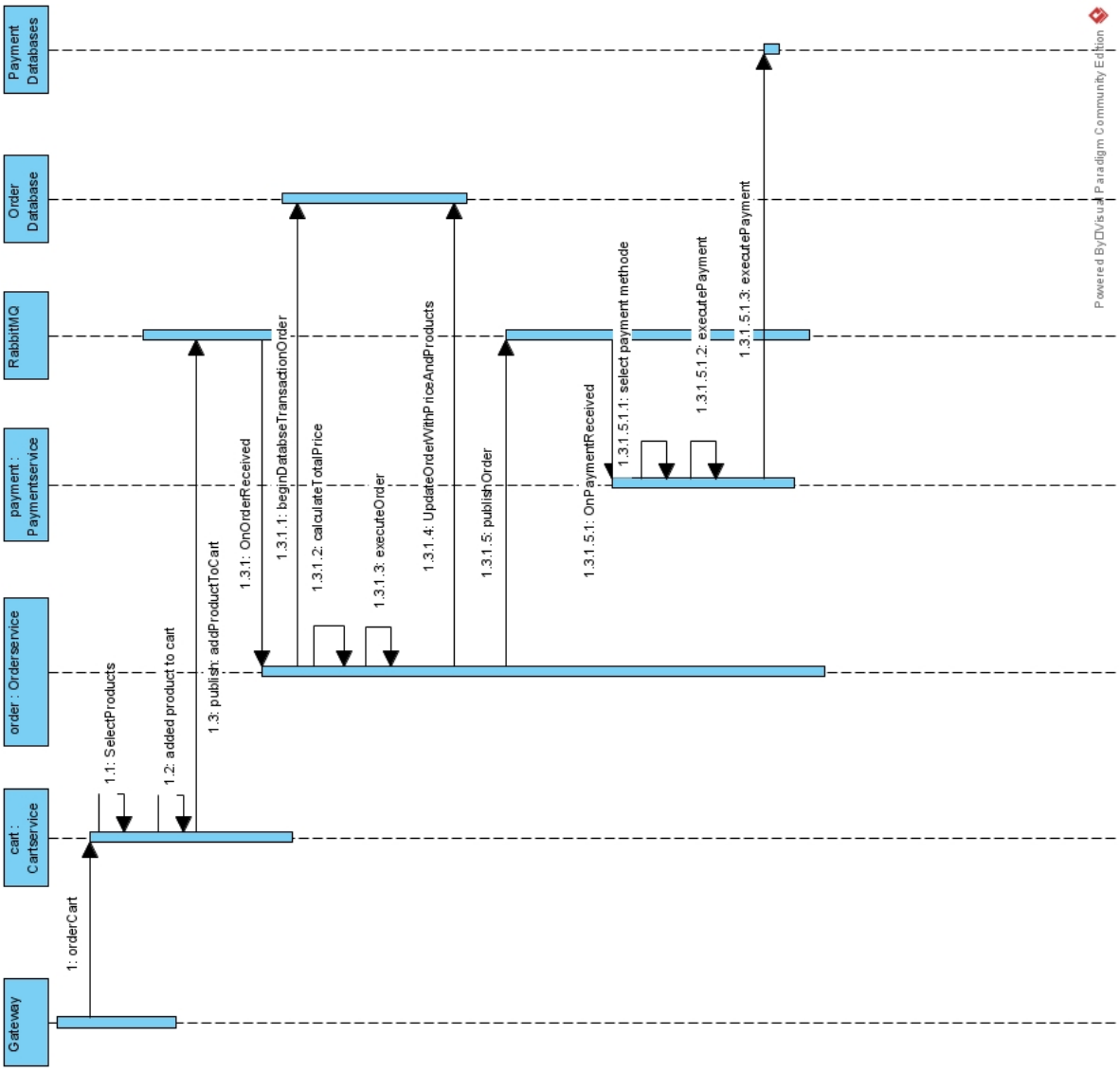


Figure 9: Sequence Diagram for Message Broker Pattern

5 Evaluation

Three prototypes are implemented by using three different patterns: Database per Service Pattern, Shared Database Pattern and Message Broker Pattern. They are tested with different size of samples in different scenarios and test results are analyzed by comparing the performances of all prototypes. Size of sample represents how many times these scenarios have been run randomly and average response time is an indicator of the performance, i.e. shorter average response time corresponding to a better performance.

5.1 Test Cases

There are three different scenarios for the system tests. These tests are implemented in JMeter and they are used for the first and the second prototypes. Part of these scenarios are designed to test the performance of these structural patterns by using REST API. As Message Broker Pattern does not use REST API for communication commands such as order_products, these scenarios are not suitable for testing for the third prototype. To test this prototype, the client is designed manually in C# and average response time is tracked to measure the performance for different size of samples. The sample sizes are 100, 200, 500, 1000 for each prototype testing. The test scenarios for first and second prototypes are the followings:

- Scenario 1:
    1. Create Customer
    2. Get All Products
    3. Get Notifications
    4. Add Notifications
    5. Dismiss Notifications
- Scenario 2:
    1. Create Customer
    2. Get All Products
    3. Order Products
    4. Get Notifications
    5. Add Notifications
    6. Dismiss Notifications
- Scenario 3:
    1. Create Customer
    2. Get All Products
    3. Add Product To Cart

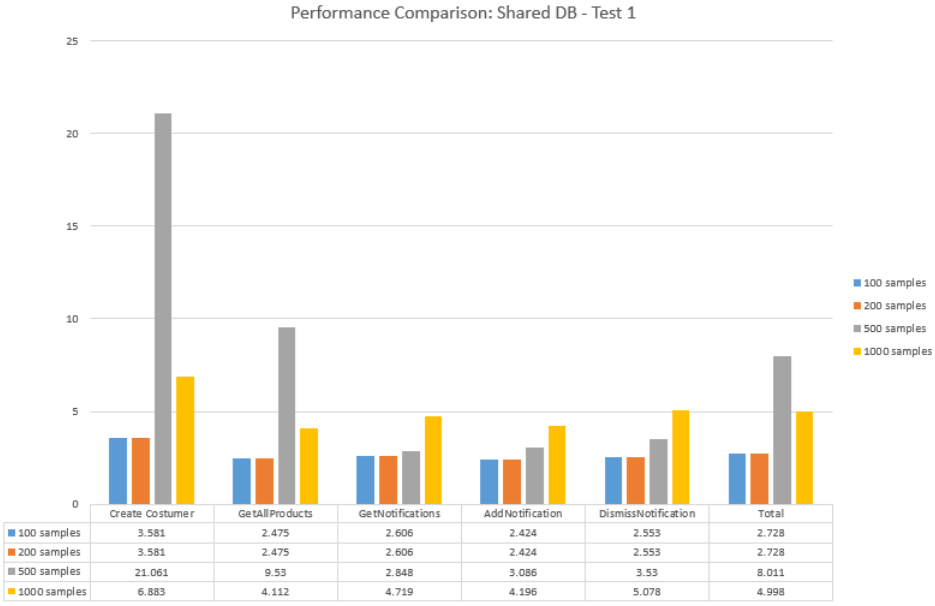
4. Get Customer Cart
5. Order Products
6. Get Notifications
7. Add Notifications
8. Dismiss Notifications

5.2 Results

The test results are summed up in the following graphs.

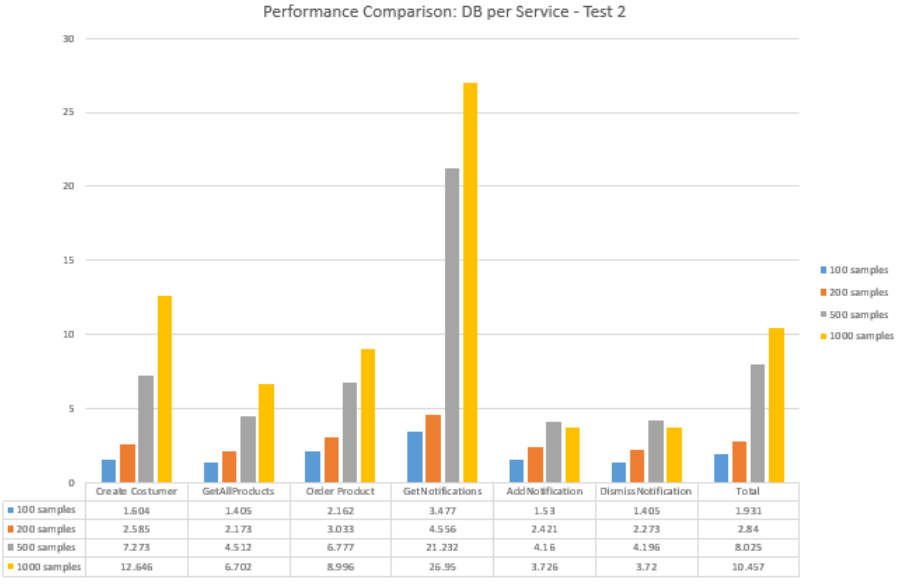
	Create Costumer	GetAllProducts	GetNotifications	AddNotification	DismissNotification	Total
100 samples	10.619	2.408	2.32	2.391	2.089	3.965
200 samples	10.619	2.408	2.32	2.391	2.089	3.965
500 samples	11.605	5.214	6.039	5.478	4.686	6.604
1000 samples	8.454	7.58	8.707	8.01	7.493	8.049

Figure 10: Test 1: DB per Service



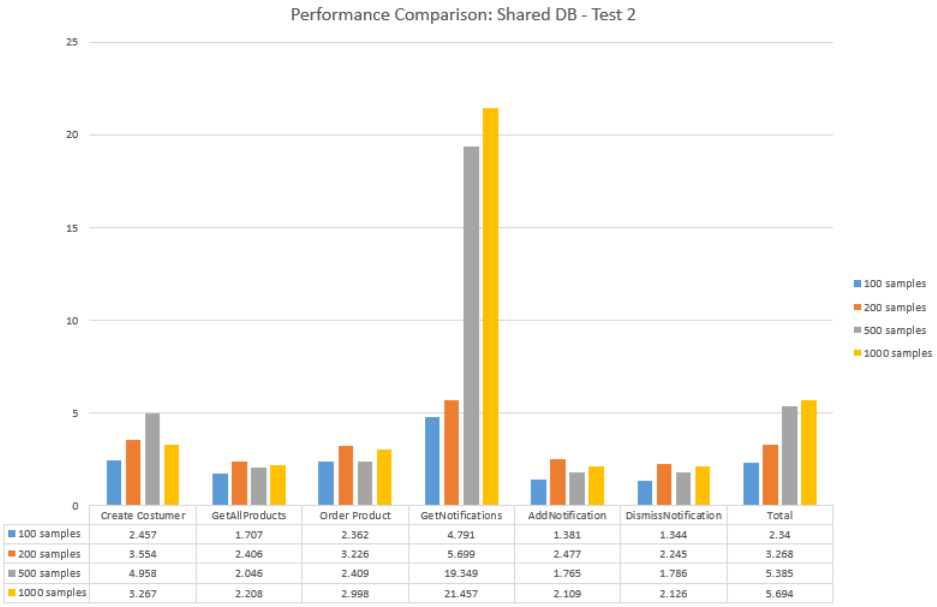
	Create Costumer	GetAllProducts	GetNotifications	AddNotification	DismissNotification	Total
100 samples	3.581	2.475	2.606	2.424	2.553	2.728
200 samples	3.581	2.475	2.606	2.424	2.553	2.728
500 samples	21.061	9.53	2.848	3.086	3.53	8.011
1000 samples	6.883	4.112	4.719	4.196	5.078	4.998

Figure 11: Test 1: Shared DB



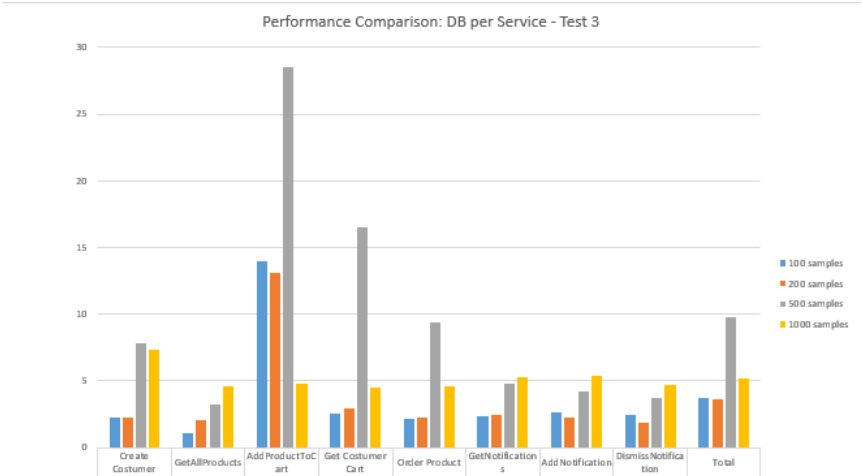
	Create Costumer	GetAllProducts	Order Product	GetNotifications	AddNotification	DismissNotification	Total
100 samples	1.604	1.405	2.162	3.477	1.53	1.405	1.931
200 samples	2.585	2.173	3.033	4.556	2.421	2.273	2.84
500 samples	7.273	4.512	6.777	21.232	4.16	4.196	8.025
1000 samples	12.646	6.702	8.996	26.95	3.726	3.72	10.457

Figure 12: Test 2: DB per Service



	Create Costumer	GetAllProducts	Order Product	GetNotifications	AddNotification	DismissNotification	Total
100 samples	2.457	1.707	2.362	4.791	1.381	1.344	2.34
200 samples	3.554	2.406	3.226	5.699	2.477	2.245	3.268
500 samples	4.958	2.046	2.409	19.349	1.765	1.786	5.385
1000 samples	3.267	2.208	2.998	21.457	2.109	2.126	5.694

Figure 13: Test 2: Shared Database

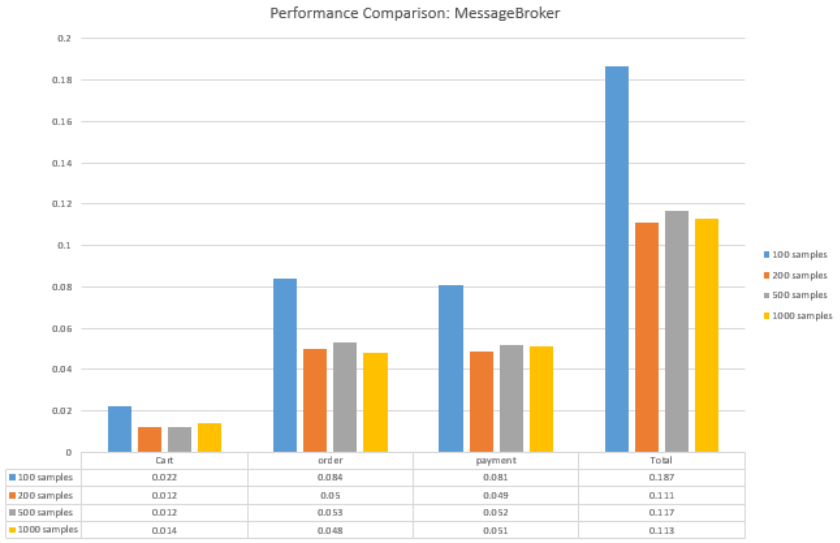


	Create Costumer	GetAllProducts	Add ProductToCart	Get Costumer Cart	Order Product	GetNotifications	Add Notification	DismissNotification	Total
100 samples	2.276	1.082	13.934	2.506	2.201	2.379	2.661	2.432	3.684
200 samples	2.238	2.088	13.114	2.894	2.252	2.436	2.302	1.909	3.654
500 samples	7.833	3.273	28.504	16.473	9.347	4.837	4.255	3.712	9.779
1000 samples	7.331	4.572	4.766	4.545	4.576	5.254	5.382	4.684	5.139

Figure 14: Test 3: DB per Service

	Create Costumer	GetAllProducts	AddProductToCart	Get Costumer Cart	Order Product	GetNotifications	AddNotification	DismissNotification	Total
100 samples	3.322	2.93	3.236	3.771	3.935	3.566	3.089	2.498	3.294
200 samples	3.93	3.231	3.32	3.63	3.56	3.713	3.604	3.139	3.516
500 samples	7.724	2.284	17.071	4.806	3.485	4.195	3.754	5.451	6.096
1000 samples	6.65	3.958	4.415	3.778	4.053	4.584	4.598	4.18	4.527

Figure 15: Test 3: Shared Database



	Cart	order	payment	Total
100 samples	0.022	0.084	0.081	0.187
200 samples	0.012	0.05	0.049	0.111
500 samples	0.012	0.053	0.052	0.117
1000 samples	0.014	0.048	0.051	0.113

Figure 16: Test: Message Broker Pattern

5.3 Discussion of the results

The test results are a good portrait of the system behaviour. Each prototype is tested with 100, 200, 500 and 1000 samples. It is found out that there has been an effect of computer hardware and performance of operating system on the outputs. Nevertheless, it is clear that the performance of the first and second prototypes get slower, when they are tested with a higher number of samples. Additionally, it is observed that when an error occurs while testing, it has a significantly slowing effect on the performance. In the graphs 14 and 15, it is clearly shown that when executing "add product to cart" operation, errors occurred in the system and the execution time was badly affected by it.

In the notification service, all current notifications are saved with their non-permanent IDs in the memory. This implementation's influence has been detected as a delay on the test results. As shown in figure 12 and 13, the average response time for getting all current notifications is greater than all other requests' average response time. When the number of samples are increased, the response time escalates accordingly.

Another fact is that in order to add a new costumer, the service needs to access to database and execute insert into query. In the results, it is shown that this request takes more time when comparing with adding a new order request. The main difference in the implementation is that database transaction happens inside of the order service. For example, as shown in the figure 12, the request for creating a costumer for 1000 samples takes 12.65 s while the request for ordering products for 1000 samples takes 9 s.

Finally, message broker pattern is used in the third prototype and its effect on increasing performance is noticeable 16. The cart service, order service and payment service are communicating via a messaging bus and in comparison with other prototypes, they do not directly communicate with each other. In this model, the asynchronous communication is used and services do not have to wait for a response anymore. This is the main reason of the increased performance.

6 Conclusion

In conclusion, all patterns used in this project have their benefits and drawbacks. However, it can be seen in the results that using an event-based communication for the microservices gets response faster in comparison with using direct communication between services. Even though database per service and shared database applications' internal structure are similar, their performances differ because of using different database architectures. In database per service, to be able to add a product to an order, it needs to join operation to both product service's and order service's databases. Meanwhile in shared database, it is enough to join these two tables. Therefore in this project, on the contrary of usual, shared database prototype works faster than database per service prototype.

7 Future Work

For the future work, it would be more convenient to use NoSQL databases for the order and product services to avoid using join operations in order to save all products inside of one order. In this particular case, it would be a better choice to use multi index feature of NoSQL databases so that all the product IDs can be saved inside of an array in the database. This array feature doesn't exist in MySQL. With this feature, it will be enough to call the products from one database instead of communicating with two databases to get product information. This service will be separated from others. Moreover, to have better and faster design, indexes can be created in the SQL database systems. Another point open for development is that implementation details of the web shop application can be improved to make the application more usable in real life.

List of Figures

1 Class Diagram for Database per Service 13
2 Component Diagram for Database per Service 14
3 Sequence Diagram for Database per Service 15
4 Class Diagram for Shared Database 17
5 Component Diagram for Shared Database 18
6 Sequence Diagram for Shared Database 19
7 Class Diagram for Message Broker Pattern 21
8 Component Diagram for Message Broker Pattern 22
9 Sequence Diagram for Message Broker Pattern 23
10 Test 1: DB per Service 25
11 Test 1: Shared DB 26
12 Test 2: DB per Service 26
13 Test 2: Shared Database 27
14 Test 3: DB per Service 27
15 Test 3: Shared Database 28
16 Test: Message Broker Pattern 28

References

[1] Microservices. https://en.wikipedia.org/wiki/Microservices, 2021.

[2] arpit jain. Three popular methods to communicate between microservices. https://medium.com/the-sixt-india-blog/microservice-communication-53cbc93cf4de, 2020.

[3] Adam Bertram. 7 reasons to switch to microservices — and 5 reasons you might not succeed. 2017.

[4] Mike Rousos Cesar de la Torre, Bill Wagner. *.NET Microservices: Architecture for Containerized .NET Applications*. Microsoft Developer Division, .NET and Visual Studio product teams, 1995.

[5] Lianping Chen. Microservices: Architecting for continuous delivery and devops. 2018.

[6] Konstantinos Plakidas Daniel Schall Fei Li Sebastian Meixner Evangelos Ntentos, Uwe Zdun. Supporting architectural decision making on data management in microservice architectures. 2017.

[7] Ihor Feoktistov. Guide to implementing microservices architecture on awss. 2021.

[8] Scott M. Fulton III. What led amazon to its own microservices architecture. 2015.

[9] Aleksandra Kwiecien. 10 companies that implemented the microservice architecture and paved the way for others. 2019.

[10] James Lewis. Microservices. 2014.

[11] Piotr Martyniuk. How to connect microservices: Part 1 types of communication. 2021.

[12] Laura Mauersberger. Why netflix, amazon, and apple care about microservices. 2017.

[13] DCtheGeek zakaria-c Youssef1313 gewarren mvelosop john-par mairaw nishanil, sughosneo. Asynchronous message-based communication. https://docs.microsoft.com/en-us/dotnet/architecture/microservices/architect-microservice-container-applications/asynchronous-message-based-communications, 2021.

[14] DCtheGeek zakaria-c Youssef1313 gewarren mvelosop john-par mairaw nishanil, sughosneo. Implementing event-based communication between microservices (integration events). https://docs.microsoft.com/en-us/dotnet/architecture/microservices/multi-container-microservice-net-applications/integration-event-based-microservice-communications, 2021.

[15] Chris Richardson. Pattern: Database per service. 2018.

[16] Chris Richardson. Pattern: Shared database. 2018.

[17] Chris Richardson. What are microservices? 2018.

[18] Alexander Schwartz. Microservices: Mehr als nur ein hype? 2017.

[19] Amazon teams. What are microservices?