Modern approaches of design software applications based on microservise architecture

Borysenko Viktor[1]
[1]Kharkiv National University of Radio Electronics, 14 Nauky Ave, Kharkiv UA-61166, Ukraine, viktor.borysenko@nure.ua

Borysenko Tatjana[2]
[2]Kharkiv National University of Radio Electronics, 14 Nauky Ave, Kharkiv UA-61166, Ukraine, tetiana.borysenko@nure.ua

Abstract. *The report covers modern approaches to improving the processes of creating Enterprise applications for complex business logic based on microservice architecture using a domain methodology.*

Keywords *design, software, Enterprise applications, Domain Driven Design, microservise architecture.*

I. INTRODUCTION AND PROBLEM STATEMENT

Microservices architecture has a positive impact on enterprise applications. Let's review general goals and principles for a microservice architecture (MSA). Here are the four goals to consider in Microservice Architecture approach [1-2]. Reduce Cost: MSA will reduce the overall cost of designing, implementing, and maintaining IT services.

Increase Release Speed: MSA will increase the speed from idea to deployment of services.

Improve Resilience: MSA will improve the resilience of our service network.

Enable Visibility: MSA support for better visibility on your service and network.

At the same time microservice architecture follow basic principles:

- Scalability;
- Availability;
- Resiliency;
- Flexibility;
- Independent, autonomous;
- Decentralized governance
- -Failure isolation;
- -Auto-Provisioning;
- -Continuous delivery through DevOps/

Business logic implements business rules is the base part of the enterprise applications. The development of an application with complex business logic is a complicated and time-consuming process.

At the same time, designing and implementation of complex business logic for applications are based on microservice architecture is harder than for monolithic applications. The main reason is the requirement to split the whole logic effectively between different microservices.

The typical domain model looks like a spiderweb of interrelated classes. To build complex software applications based on the microservice architecture it is required to solve two essential problems.

The first problem is that hierarchy of classes in microservice architecture should be split by services, unlike monolithic architecture. Therefore, first of all, it is necessary to get rid of the objects' references that cross boundaries of services.

The second problem lies in the design of the business logic that is restricted by the usage of transactions in a microservice architecture.

II. PROBLEM SOLUTION AND RESULTS

The current work uses the modern methodology of domain-driven design (DDD) as a fundamental approach for developing enterprise applications [3]. This approach includes firstly usage of strategic and secondarily tactic design patterns.

Basic concepts of domain-driven design using strategic templates [4]:

- Single language;
- Limited context;
- Subject domain,
- Subject subdomain;
- Semantic core;
- Context map.

Strategic design patterns are used in different modern enterprise applications as building blocks. Some of them are supported by such frameworks as JPA and Spring. To achieve strategic development is enough to use such tools.

It is proposed to use the base pattern "Aggregate" in this work that is one of tactical design patterns used in DDD methodology. It structures the business logic as a set of aggregates. These building blocks are very useful during development of microservices.

The domain model describes a set of classes and the relationship between them in traditional object-oriented design. Classes are usually grouped into packages. The boundaries between different business objects are not clear in the traditional domain model. Such ambiguous vague separation may cause problems, especially in microservice architecture.

The lack of clear boundaries also causes problems when updating a business object in addition to conceptual uncertainty. A typical business object has invariants, i.e. special business rules that must always be followed. But for observance of invariants it is necessary to carefully design business logic.

Changing or updating parts of a business object directly may result in violation of business rules. "Aggregate" as tactical pattern of DDD methodology helps to solve this problem effectively.

In this case, the aggregate is the cluster of domain objects that can be used as unified whole. It consists of a root entity, as well as one or more entities and objects. Many business objects are designed as aggregates. For example, "Gas transportation" subject domain contains some nouns like "Gas pipeline section", "Compressor yard ", "Compressor station" are aggregates.

The “Aggregate” pattern creates a business model in the form of a set of aggregates, i.e. graphs of objects that can be used as a unified whole. Structuring the domain model as a set of aggregates defines clear boundaries.

Aggregates break down the domain model into blocks and it’s easier to design them individually. They also determine the scope of operations, such as updating, fetching, and deleting.

The aggregate is often loaded from the entire database, it allows to avoids any problems with lazy loading.

When an aggregate is deleted from the database all its objects are deleted too.

Updating the whole aggregate, and not its individual parts, solves problems with consistency as described in the previous example. Update operations are called for the root of the aggregate, which ensures the observance of invariants.

In addition, in order to maintain competitiveness, the aggregate root is blocked by version number or database isolation level. However, it should be mentioned that this approach does not require updating the entire aggregate in the database.

Another rule that aggregates must obey is that a transaction can only create or update one aggregate. This limitation is ideal for microservice architecture. It ensures that the transaction does not overstep the limits of the service. It also agrees well with the limited transnational model of most NoSQL databases.

It is important to decide how big it is necessary to make this or that aggregate during developing a domain model. On the one hand, ideally, aggregates should be small.

This will increase the number of simultaneous requests that your application is able ot handle and improve scalability as each aggregate's updates are serialized.

This will also have a positive effect on the experience of interaction, as it reduces the probability that two users will try to make conflicting changes to the same aggregate.

But on the other hand, an aggregate is the scope of a transaction, therefore, in order to ensure the atomicity of a certain update, on the contrary it is worth making it larger.

The negative aspect of large aggregates in the context of microservice architecture is that they prevent decomposition. For example, the business logic for orders and customers should be in the same service, which makes this service more volumetric. Considering these problems it is better to make aggregates as small as possible.

The main part of the business logic consists of aggregates in a standard microservice. The rest of the code belongs to domain services and narratives.

Narratives orchestrate local transaction chains to ensure data consistency.

Services serve as entry points of business logic and are called by inbound adapters.

The service uses the repository to retrieve aggregates or save them to the database.

Each repository is implemented by an outgoing adapter that accesses the database.

In the context of DDD, a domain event is something that happened with an aggregate.

In a domain model it is a class. An event usually represents a state change. In this work, it is recommended to use the “Domain Event” template - the aggregate publishes a domain event at the time of its creation or during some other significant change.

The usefulness of domain events relates to the fact that other parts of the interaction (users, external applications, or other components within the same application) are often interested in information about changes in the state of the aggregate.

A domain event is a class with a name based on the passive participle of the past tense. It contains properties that expressively describe this event. Each property is either a simple value or an object.

A domain event usually has metadata, such as its identifier and timestamp. It may carry the identifier of the user who made the change, as far as it is useful for audit. Metadata can be part of an event object - possibly defined in the parent class. Or they can be inside the wrapper around the event object. The identifier of the aggregate that generates the event may also not be its direct property, but it can be part of the wrapper.

But the disadvantage of requesting an aggregate from a service is the additional costs of fulfilling this request. Alternatively, you can use event enrichment.

It means that events contain the information that a consumer needs. As a result, event consumers become simpler because they no longer need to request data from the service that posted the event. Event enrichment simplifies consumers, but the drawback of such approach is the risk of violation of open/closed SOLID principle for event classes.

These classes can potentially be changed each time when clients’ requirements are changed. This can adversely affect support of event as such kind of changes can affect several parts of the application.

Earlier, the main reasons why aggregates are suitable for developing business logic in a microservice architecture were presented.

When an aggregate is created or updated it must publish domain events. These events have many implementation areas. Subscribers of domain events notify users and other applications, as well as publish messages in a client browser via WebSocket.

III. CONCLUSIONS

A good way to organize the business logic of a microservice is to split it into aggregates according to the DDD principle. Aggregates make the domain model more modular, exclude the possibility of using object references between services and ensure that each ACID transaction is performed within the same service.

REFERENCES

[1] Irakli Nadareishvili, Matt McLarty, Michael Amundsen/ Microservice Architecture: Aligning Principles, Practices, and Culture/- O'Reilly, 2016.-144

[2] Kasun Indrasiri, Prabath Siriwardena. Microservices for the Enterprise.- Apress, 2018.- 434 p.

[3] Vijay Nair. Practical Domain-Driven Design in Enterprise Java - Using Jakarta EE, Eclipse MicroProfile, Spring Boot, and the Axon Framework.-Apress, 2019.- 388

[4] Chris Richardson. Microservices Patterns: With examples in Java. Manning Publications: 2018.- 522Felipe Gutierrez. Introducing Spring Framework.-Apress:2014.-352