DE Store Architecture Proposal

Software Architecture | SET10401 2023-4 TR3

Matthew Jack

Matriculation No. 40509169

Word Count: 4011

Contents

Introduction

Choosing the right software architecture is key for developing a system that is not only functional but also scalable and maintainable over time. The DE-Store project aims to create a distributed business management system with some of the core features including price control, inventory management, loyalty programs, finance approval, and performance analysis.

Given the distributed nature of the DE-Store system, two architectural approaches were considered: a client-server monolithic architecture and a Service-Oriented Architecture (SOA). The client-server monolithic architecture involves a simpler design where all components are tightly integrated within a single application. This approach can be beneficial for rapid development and straightforward deployment. However, as the system expands, a monolithic structure can present challenges in terms of scalability, flexibility, and the ability to integrate with other systems (Lewis & Fowler, n.d.).

On the other hand, SOA offers a more modular approach where individual services are responsible for specific business functions. These services are independent and can communicate with each other over a network. This modularity makes SOA a more flexible and scalable option, particularly for distributed systems like DE-Store, where different functionalities may need to evolve independently or integrate with external services.

This report will first compare the client-server monolithic architecture and SOA, highlighting the advantages and drawbacks of each. It will then explain the decision to adopt SOA for the DE-Store system and provide a detailed overview of the system's design and evaluation based on this architectural choice.

Architectural Options Considered

Client-Server Monolithic Architecture

The client-server monolithic architecture is a traditional software design approach where the entire application is built as a single, cohesive unit. This architecture is characterised by a strong dependency between the client and server, where the server is responsible for handling all of the business logic, data processing, and interactions with the underlying database.

Components:

- **Client:** The client is the front-end component of the system, typically responsible for presenting the user interface and managing user interactions. In this architecture, the client relies heavily on the server for data and application functionality, sending requests to the server for processing and receiving responses to update the user interface.
- **Server:** The server is the core of the monolithic architecture, encompassing all business logic, data processing, and integration with the database. The server handles requests from clients, performs the necessary operations (such as querying or updating data), and returns the results to the client. All components and functionalities of the application are contained within this single server unit.
- **Database:** The system typically uses a single, centralised database. This database stores all the data required by the application, and the server interacts with it to perform CRUD (Create, Read, Update, Delete) operations.

Connectors:

- **Client-Server Communication:** The client and server communicate over a network, typically using a standard protocol like HTTP/HTTPS. The client sends requests to the server, which processes these requests and returns the necessary data or responses.
- **Server-Database Interaction:** The server directly interacts with the database to retrieve and manipulate data. This interaction is tightly integrated into the server's codebase, meaning all database operations are managed and controlled by the server.

Information Exchange: In a client-server monolithic architecture, all the communication and information exchange occur through the server. The client sends requests for specific actions or data, the server processes these requests by applying business logic and interacting with the database, and then the server returns the required information to the client. The entire process is managed within the confines of a single application running on the server.

Pros and Cons:

- **Advantages:**
    - **Simplicity:** The monolithic architecture is straightforward to design, develop, and deploy. All the application’s components are integrated into a single unit, simplifying the development process and reducing the complexity of managing multiple services or components.
    - **Unified Codebase:** With a single codebase, it is easier to maintain consistency across the application, as all functionality is contained within one system. This can streamline development and make it easier to manage changes.
    - **Ease of Deployment:** Deployment is simpler in a monolithic architecture since the entire application is packaged and deployed as a single unit, reducing the complexity of deploying and managing multiple interdependent components.
- **Disadvantages:**
    - **Scalability Issues:** As the application grows, the monolithic architecture can become cumbersome and difficult to scale. Scaling typically involves duplicating the entire application, which can lead to inefficiencies and increased resource usage.
    - **Tight Coupling:** The components within a monolithic architecture are tightly coupled, meaning that changes to one part of the system can impact other parts. This can lead to increased complexity and a higher likelihood of introducing bugs when making changes or updates.
    - **Limited Flexibility:** The monolithic nature of the architecture can limit the ability to integrate new technologies or make architectural changes. Introducing new features or integrating with external systems often requires significant modifications to the entire application, making it less flexible in adapting to change.

The client-server monolithic architecture provides a cohesive, single-unit design that works well for small to medium-sized applications with straightforward requirements. However, as applications grow in complexity and scale, this architecture can present challenges in terms of scalability, maintainability, and flexibility, especially for distributed systems like DE-Store.

Service-Oriented Architecture (SOA)

Service-Oriented Architecture (SOA) is an architectural approach that organises a system as a collection of independent, loosely coupled services. Each service is designed to perform a specific business function and communicates with other services over a network. SOA aims to provide flexibility, scalability, and reusability by allowing services to be developed, deployed, and maintained independently while still functioning as part of a larger system (Erl, 2005).

Components:

- **Services:** In SOA, the system is broken down into discrete services, each responsible for a specific business function. For example, there might be separate services for price control, inventory management, loyalty programs, finance approval, and performance analysis. Each service is autonomous and encapsulates the business logic related to its function.
- **Service Consumers:** These are the clients or other services that consume the services provided by the system. Service consumers can be user interfaces, other services, or external systems that interact with the services to fulfil specific tasks.
- **Service Registry:** A service registry is an optional component in SOA that keeps track of available services and their locations. It allows service consumers to discover and connect to the services they need dynamically.
- **Message Broker (Optional):** In more complex SOA implementations, a message broker may be used to facilitate asynchronous communication between services, allowing them to send and receive messages without direct dependencies.
- **External Systems Integration:** SOA is well-suited for integrating with external systems. For example, in DE-Store, a service might interact with an external finance approval system, treating it as just another service within the architecture.

Connectors:

- **Service Communication:** Services in SOA communicate with each other and with service consumers over a network using standardised protocols such as HTTP/HTTPS, SOAP, or REST. Each service exposes a well-defined API that other services or clients can use to interact with it.
- **Loose Coupling:** One of the key principles of SOA is loose coupling, where services are designed to be independent of each other. This allows services to be updated, replaced, or scaled independently without affecting the overall system.

Information Exchange: In SOA, information exchange happens through service-to-service communication or between service consumers and services. Each service typically operates with its own data store or database, allowing for a more modular approach to data management. Services can communicate synchronously, where one service waits for a response from another, or asynchronously, where messages are sent and processed independently of the immediate response.

Pros and Cons:

- **Advantages:**
  - **Scalability:** SOA allows for individual services to be scaled independently based on demand. This makes it easier to handle varying loads on different parts of the system without scaling the entire application.
  - **Modularity:** By breaking down the system into independent services, SOA promotes modularity, making it easier to develop, test, deploy, and maintain different parts of the system.
  - **Reusability:** Services in SOA can be reused across different applications or systems, reducing duplication of effort and enabling better resource utilisation.
  - **Flexibility:** SOA is highly flexible, allowing services to be developed using different technologies or programming languages, as long as they adhere to the agreed-upon communication protocols. This flexibility extends to integrating with external systems, making SOA a strong choice for distributed and heterogeneous environments.
- **Disadvantages:**
  - **Complexity:** SOA introduces additional complexity compared to a monolithic architecture. Managing multiple services, each with its own API, can be challenging, especially when dealing with service discovery, security, and inter-service communication.
  - **Overhead:** The need for communication between services, often over a network, introduces latency and overhead. This can impact performance, particularly in systems requiring real-time processing.
  - **Dependency Management:** While services are loosely coupled, they may still depend on each other to complete business processes. Coordinating these dependencies and ensuring data consistency across services can be complex.

Service-Oriented Architecture offers a more modular, flexible, and scalable approach compared to traditional monolithic architectures. It is particularly well-suited for distributed systems like DE-Store, where different parts of the system may need to evolve independently or integrate with external services. However, the benefits of SOA come with the trade-offs of increased complexity and potential performance overhead, which must be managed carefully to realise the full potential of the architecture.

Architecture Selection Rationale

When evaluating the architectural options for the DE-Store system, scalability was seen as a critical factor in determining the most suitable approach. Given the distributed nature of the DE-Store system and its potential for growth, the ability to scale efficiently and effectively was a key consideration.

Scalability Challenges with the Monolithic Architecture

The client-server monolithic architecture, while straightforward and easier to implement initially, presents significant challenges when it comes to scalability. In a monolithic system, all components—business logic, data processing, and database interactions—are tightly integrated into a single application. This architecture typically scales **vertically**, meaning that to handle increased load, the entire application must be moved to a more powerful server or hardware must be upgraded. While this approach can work in the short term, it quickly becomes limited.

Vertical scaling has inherent constraints:

- **Hardware Limits:** There is a physical limit to how much a single server can be upgraded in terms of CPU, memory, and storage. As the system grows, these limits are quickly reached, leading to performance bottlenecks.
- **Cost Inefficiency:** Upgrading to more powerful hardware can be expensive, and the cost increases exponentially as the system scales. This approach also lacks flexibility, as the entire system is dependent on the performance of a single machine.
- **Single Point of Failure:** A monolithic architecture places the entire system on a single server or a cluster of servers, making it more vulnerable to failures. If the server goes down, the entire application becomes unavailable.

Once vertical scaling is no longer feasible, the next step is **horizontal scaling**, which involves distributing the load across multiple servers. However, monolithic architectures are not inherently designed for horizontal scaling, making this transition complex and challenging:

- **Complex Refactoring:** To enable horizontal scaling, the monolithic application must be refactored to distribute its components across multiple servers. This often involves significant code changes and restructuring, which can be time-consuming and error prone.

- **Operational Complexity:** Managing a horizontally scaled monolithic application introduces additional complexity in terms of load balancing, data consistency, and deployment. These complexities are difficult to address retrospectively, especially when the system was not originally designed with horizontal scaling in mind.

Advantages of Service-Oriented Architecture (SOA) in Scaling

Service-Oriented Architecture (SOA) addresses these scalability challenges more effectively by allowing the system to scale **horizontally** from the outset. In SOA, the system is composed of independent services, each responsible for a specific business function. These services can be deployed, managed, and scaled separately, providing several key advantages:

- **Horizontal Scalability:** Unlike a monolithic architecture, SOA is designed to support horizontal scaling from the beginning. Each service can be scaled independently by deploying additional instances of that service across multiple servers. This approach allows the system to handle increased load more effectively and flexibly.
- **Targeted Resource Allocation:** In SOA, resources can be allocated based on the specific needs of each service. For example, if the inventory management service experiences higher traffic, only that service needs to be scaled, rather than the entire system. This targeted approach is more efficient and cost-effective.
- **Avoiding Single Points of Failure:** Because services are distributed across multiple servers, SOA reduces the risk of a single point of failure. If one service or server goes down, other services can continue to operate, ensuring higher availability and reliability of the system.
- **Simplified Maintenance and Evolution:** With SOA, individual services can be updated, replaced, or scaled without affecting the rest of the system. This modularity simplifies maintenance and allows the system to evolve over time, incorporating new technologies or adapting to changing business needs without extensive refactoring.
- **Early Investment in Complexity Pays Off:** While SOA is more complex to implement initially, requiring careful design of service boundaries, APIs, and inter-service communication, this upfront complexity pays off in the long run. As the DE-Store system grows, the ability to scale services independently and manage them as separate entities ensures that the system remains manageable, flexible, and scalable.

Given the potential for growth and the distributed nature of the DE-Store system, the limitations of the monolithic architecture, particularly in terms of scalability, make it a less suitable choice. The challenges associated with scaling a monolithic system, especially the transition from vertical to horizontal scaling, would introduce significant complexity and costs as the system evolves.

In comparison, Service-Oriented Architecture (SOA) offers a scalable and flexible solution that aligns with the needs of the DE-Store system. By allowing for independent scaling of services and reducing the risks associated with single points of failure, SOA provides a robust foundation for building a system that can grow and adapt to future demands. This makes SOA the more appropriate choice for DE-Store, ensuring that the system can meet current requirements while also being well-prepared for future expansion.

System Design

The system design for DE-Store leverages a Service-Oriented Architecture (SOA) that is optimised for scalability, flexibility, and maintainability. The design includes a web-based architecture with load balancing, an API Gateway, and a centralized Next.js frontend. While the system starts with a central database, it is designed to allow for future subdivision, or the addition of smaller, service-specific databases as needed.

High-Level Architecture Overview

The DE-Store system is composed of several key components that work together to provide a scalable and efficient distributed system. The main components of the architecture are:

- **API Gateway:** Acts as a single-entry point for all client requests. It routes requests to the appropriate service, handles cross-cutting concerns like authentication, rate limiting, and load balancing, and aggregates responses when necessary.
- **Business Services:** Independent services that handle specific business functions such as Authentication, Customer Management, Product Management, Analytics, and Store Management. Each service is designed to be loosely coupled and independently deployable.
- **Load Balancer:** In the recommended configuration, a load balancer is positioned after the API Gateway, distributing traffic among multiple instances of each service. This ensures that no single service instance becomes overwhelmed, improving reliability and performance. If required, an additional load balancer could be placed before the API Gateway in the future, to distribute incoming traffic across multiple API Gateway instances. This dual load balancer approach would be beneficial in scenarios with very high traffic volumes, ensuring that both the API Gateway and backend services remain highly available and scalable.
- **Next.js Frontend:** A centralised frontend application built with Next.js, responsible for presenting the user interface and interacting with the backend services through the API Gateway. While micro frontends could be adopted in the future if needed, the current approach focuses on a single, cohesive frontend application to reduce complexity. Using a web-based framework like Next JS also has the benefit of being highly portable since it can run in any

browser removing the need for separate codebases to support each native platform like Windows or Android.

- **Central Database:** Initially, a single central database is used to store data for all services. This database may be subdivided into schemas or split into smaller, service-specific databases in the future, depending on evolving requirements and performance needs.
- **External Systems Integration:** The architecture is designed to easily integrate with external systems, ensuring that services can interact with third-party applications as needed.

Diagram: High-Level Architecture

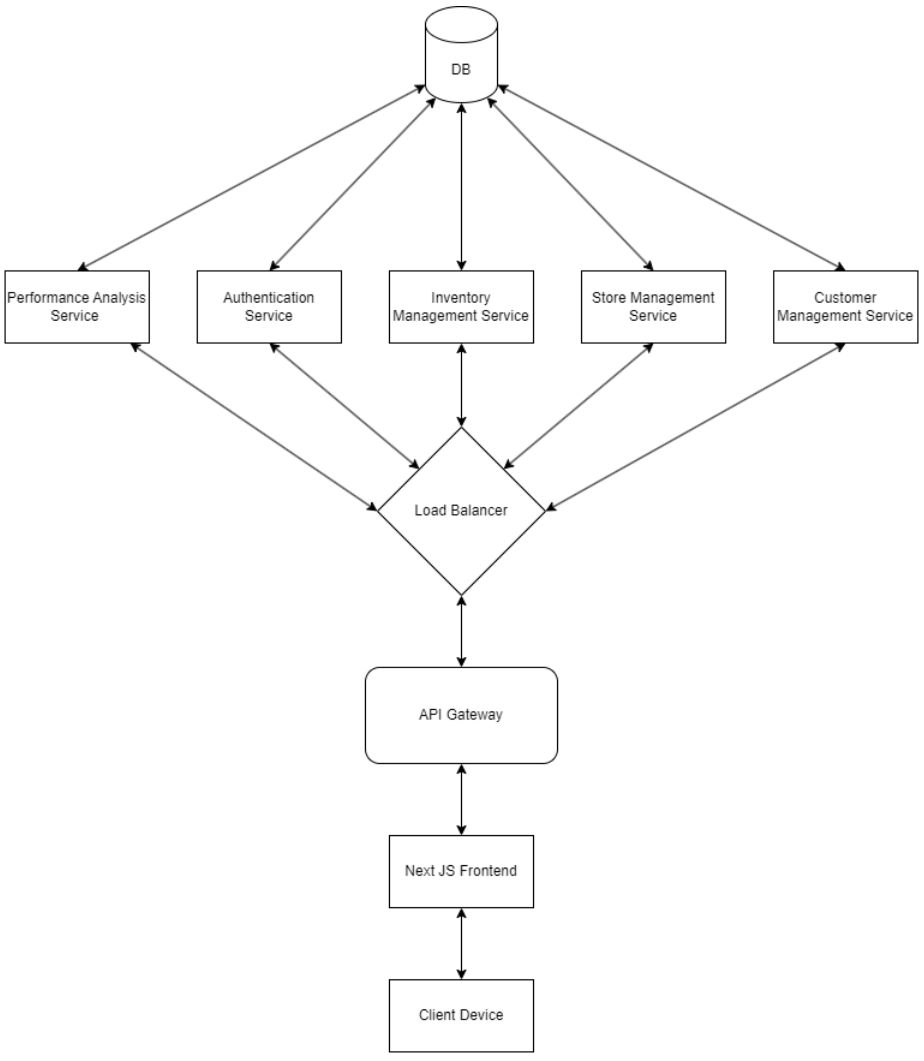

Figure 1 Diagram example of overall architecture with one load balancer

API Gateway

The API Gateway is a key component in the SOA for DE-Store, serving as the interface between the frontend application and the backend services (Newman, 2015). It provides several important functions:

- **Request Routing:** The API Gateway routes incoming client requests to the appropriate service based on the URL path or other request parameters. This ensures that each request reaches the correct service without exposing the internal architecture to the client.
- **Cross-Cutting Concerns:** The Gateway handles cross-cutting concerns such as authentication, rate limiting, and logging. This centralisation of common concerns simplifies the development of individual services by offloading these responsibilities to the Gateway.
- **Load Balancing:** The API Gateway works in conjunction with the load balancer to distribute requests evenly across multiple instances of each service, enhancing the system's capacity to handle high traffic volumes.

Business Services Design

Each business function in DE-Store is encapsulated within its own service, which operates independently from the others. The design principles for these services include:

- **Loose Coupling:** Services are designed to operate independently, with well-defined APIs that allow them to interact without requiring direct dependencies. This reduces the risk of cascading failures and allows services to be developed, tested, and deployed independently.
- **Centralised Database with Potential for Subdivision:** Initially, all services share a single, centralised database. However, the design allows for future subdivision into separate schemas or the use of smaller, service-specific databases, especially for data that does not need to be accessed by other services.
- **Scalability:** Services are designed to be horizontally scalable, meaning that additional instances of a service can be deployed to handle increased load. The load balancer and API Gateway work together to distribute traffic across these instances.

Key Services:

- **Authentication Service:** Manages user authentication and authorisation, ensuring secure access to the system.
- **Customer Management Service:** Handles customer-related data and processes, including customer profiles, accounts, and interactions.
- **Product Management Service:** Manages product data, pricing, and availability within the store.
- **Analytics Service:** Gathers and processes data for reporting and performance analysis across the system.
- **Store Management Service:** Oversees store operations, including inventory management, sales tracking, and other operational tasks.

Load Balancing

Load balancing is critical to ensuring that the DE-Store system can handle high traffic volumes without degrading performance. The load balancer distributes incoming requests from the API Gateway across multiple instances of each service, preventing any single instance from becoming overwhelmed.

- **Horizontal Scaling:** Load balancing supports horizontal scaling by allowing more instances of a service to be added dynamically in response to increased demand. This helps maintain high availability and performance during peak usage times.
- **Failover Support:** In case of a service instance failure, the load balancer automatically reroutes traffic to other available instances, minimising downtime and ensuring that the system remains operational.

Frontend Design (Next.js)

The frontend of DE-Store is a single application built using Next.js. This approach simplifies the development process by keeping the frontend unified, while still allowing for potential expansion into micro frontends in the future if needed.

- **Server-Side Rendering (SSR):** Next.js enables server-side rendering, which improves the performance of the frontend by generating the HTML on the server before sending it to the client. This is particularly beneficial for providing a faster initial load time.
- **API Integration:** The frontend interacts with the backend services exclusively through the API Gateway. This abstraction ensures that the frontend remains decoupled from the complexities of the backend architecture, allowing it to focus on presenting data and managing user interactions.
- **Potential for Micro Frontends:** While micro frontends offer modularity and independent deployment, they introduce significant complexity compared to backend services (Geers, 2020). Decoupling the UI can lead to inconsistencies in design and user experience, increased interdependencies, and higher maintenance overhead. Each micro frontend requires careful coordination of shared components, routing, and state management, which can slow development and introduce performance issues. For these reasons, the current architecture favours a single Next.js frontend, with the option to adopt micro frontends later, using tools like Webpacks Module Federation plugin, if the system's complexity necessitates it.

Data Management

The DE-Store system starts with a centralised database that serves all the business services. This setup simplifies initial development and deployment, while still providing the flexibility to evolve as needed:

- **Centralised Database:** All services initially use a single, centralised database, ensuring that data management is consistent and straightforward.

- **Potential for Subdivision:** The architecture is designed to allow for future subdivision of the database into schemas specific to each service, or the introduction of smaller, service-specific databases. This approach will be particularly useful for services that manage data not accessed by others, reducing the risk of inter-service coupling through shared data.
- **Data Consistency:** With a centralised database, maintaining data consistency across services is more straightforward. If the database is subdivided or split in the future, the system will support eventual consistency across services using event-driven mechanisms or distributed transactions.

External Systems Integration

DE-Store's architecture is designed to easily integrate with external systems. These integrations are handled through well-defined APIs, ensuring that services can interact with third-party applications as needed.

- **API-Based Integration:** For example, the Analytics Service or the Store Management Service might interact with external systems for additional data processing or third-party analytics tools.
- **Loose Coupling:** The use of APIs ensures that integrations with external systems are loosely coupled, reducing the risk of disruptions in DE-Store if the external systems experience issues.

Conclusion

The system design for DE-Store leverages the benefits of a Service-Oriented Architecture, coupled with modern web-based components like load balancing and an API Gateway. By adopting SOA with a centralised database (with flexibility for future subdivision), the system is well-positioned to scale efficiently, manage complexity, and integrate well with external systems, all while maintaining a centralised, easy-to-manage frontend. This design ensures that DE-Store can meet its current business needs while remaining adaptable to future demands.

System Evaluation

The evaluation of the DE-Store system focuses on scalability, flexibility, maintainability, and overall system performance. These criteria are essential to ensure that the chosen architecture—Service-Oriented Architecture (SOA) with a centralised database and an API Gateway—meets the current and future needs of the DE-Store system.

Scalability

The SOA design allows the DE-Store system to scale horizontally by adding more instances of each service as demand increases, with load balancing ensuring that no single service instance becomes a bottleneck. Initially, a central database is used, but the architecture allows for future subdivision into service-specific schemas or databases, enhancing scalability as the system grows.

Flexibility

The SOA design promotes flexibility through modular services, each independently deployable and updatable. The API Gateway simplifies routing, authentication, and integration with external systems, ensuring the system can easily adapt to evolving business requirements or third-party integrations.

Maintainability

SOA's separation of concerns leads to a more maintainable codebase, with each service focused on a specific business function. While the centralised database simplifies initial data management, the architecture supports a transition to service-specific databases if needed, enhancing maintainability by reducing inter-service dependencies.

System Performance

The API Gateway and load balancing ensure the system can handle concurrent requests efficiently, preventing performance bottlenecks. As traffic increases, the architecture allows for scaling and database subdivision to maintain optimal performance.

Final Conclusion

The DE-Store system, built using a Service-Oriented Architecture, provides a scalable, flexible, and maintainable foundation for a distributed business management system. The architecture's modularity and the ability to evolve the database design ensure the system can meet current operational needs while remaining adaptable to future growth. The inclusion of an API Gateway and load balancing further supports the system's ability to handle increased traffic and integrate with external services. This design positions DE-Store to succeed both now and in the future, balancing simplicity with the capacity for expansion.

References

Erl, T. (2005). *Service-oriented architecture: Concepts, technology, and design*. Prentice Hall.

Geers, M. (2020). *Micro frontends in action*. Manning Publications.

Lewis, J., & Fowler, M. (n.d.). *Monolithic vs microservices architecture: How to choose*. Retrieved from https://martinfowler.com/articles/microservices.html

Newman, S. (2015). *Building microservices: Designing fine-grained systems*. O'Reilly Media.