Christoffer Perch Nielsen & Jonathan Gjetting Braad

System Integrations - OLA 1

Technology Stack

For this semester, we have selected a technology stack that will facilitate the development, collaboration, and integration of our projects.

Version Control Platform:

We will be using **Git** on **GitHub** as our version control system. GitHub is familiar to our team, and its features for collaboration, branch management, and code review align well with our workflow, ensuring smooth project management and version tracking.

Text Editing:

For group collaboration and easy document sharing, we will primarily use **Google Docs**. This allows for real-time collaboration, especially useful for the early stages of our research. As our documentation grows, we plan to transition to tools like **LaTeX** for larger reports and technical documents. Additionally, we will create a **README** file in Markdown for project documentation, ensuring clear instructions and descriptions.

Online Research Tools:

We will utilize both **Steven's provided materials** and online tools like **Google** and **ChatGPT** to gather relevant information. These tools will help us explore enterprise integration concepts, trends, and best practices.

Development Stack:

- **Programming Language**: We have chosen **C#** for its robust support within our selected framework and its familiarity. We will also incorporate **HTML, CSS, and JavaScript** for front-end development.
- **Framework**: We will primarily use **ASP.NET**, a powerful web application framework that integrates well with C# and supports the development of dynamic, scalable web applications.
- **Database**: Our primary database will be **Microsoft SQL Server (MSSQL)** for its reliability and integration with ASP.NET. If additional database requirements arise, we will explore other databases we learned in our first semester, such as MongoDB.

Middleware Tools:

In addition to **REST APIs** for communication between our services, we are evaluating middleware options such as **RabbitMQ** and **Apache Kafka** to handle messaging and asynchronous data processing. We will decide based on project needs as they arise.

Enterprise Integration

Enterprise Integration refers to the process of ensuring different systems, services, and applications within an organization work together efficiently. It involves creating communication layers between services (whether monolithic or microservices) to allow them to exchange data seamlessly.
Popular integration methods include:

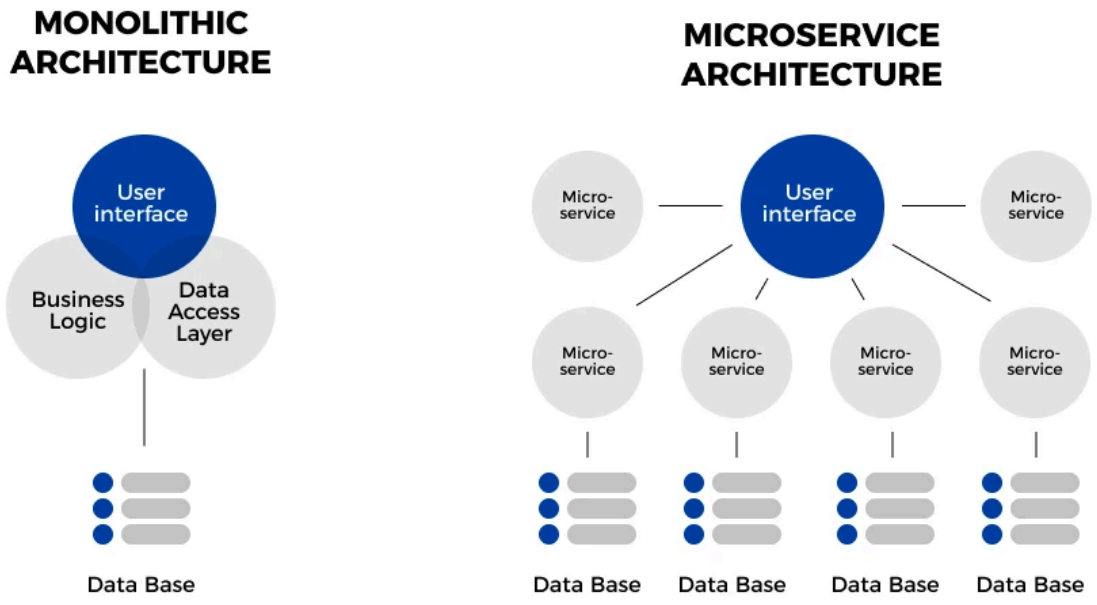
- **Message brokering** (e.g., RabbitMQ, Kafka)
- **Service Oriented Architecture** (SOA)
- **Microservices architecture** that decouples services into smaller, independent units.

Technologies for Enterprise Integration:
Enterprise integration relies on key technologies to ensure seamless communication between services. Message brokers like RabbitMQ and Apache Kafka handle asynchronous messaging, allowing decoupled communication in distributed systems. API Gateways such as Kong or AWS API Gateway manage traffic and security between services. For internal integration, frameworks like Apache Camel and Spring Integration implement patterns like Pipes and Filters, making it easier to connect different systems. Additionally, orchestration tools like Kubernetes manage deployment and scaling, ensuring efficient service interaction in complex environments.

Diagramming Standards:

Diagramming standards are essential for visualizing complex system architectures and workflows in enterprise integration. UML (Unified Modeling Language) is commonly used to create system and interaction diagrams, helping to illustrate relationships between components. BPMN (Business Process Model and Notation) is another standard focused on mapping business workflows, making it ideal for process integration. Tools like Mermaid and Miro are popular for creating flowcharts and sequence diagrams, offering a simple way to represent integration patterns and system designs in a collaborative, visual format.

Monolithic vs Microservices:



Monolithic Applications: Single, unified codebase where all components are tightly coupled. It may still use internal integration patterns.

Microservices: Distributed systems where services are independent, loosely coupled, and often communicate through APIs or message brokers.

While both monolithic and microservices architectures are viable options for building applications, they differ significantly in terms of scalability, flexibility, and complexity.

Monolithic applications are simpler to develop initially because they operate as a single, unified codebase where all components (such as the UI, business logic, and database) are tightly coupled. This makes deployment easier but can lead to challenges as the application grows. Updating or scaling specific parts of a monolithic system often requires redeploying the entire application, which can increase downtime and complexity.

On the other hand, **microservices architectures** divide an application into small, independent services that can be developed, deployed, and scaled separately. Each service is loosely coupled and typically communicates via APIs or message brokers. This allows for greater flexibility, as teams can work on different services independently, and scaling can be done on a per-service basis. However, microservices introduce complexity in areas such as communication between services, data consistency, and deployment, requiring more sophisticated orchestration tools and patterns like event-driven architecture.

In essence, monolithic systems are simpler to manage in the early stages but may become rigid as they scale, while microservices offer long-term scalability and flexibility at the cost of increased complexity.

Integration Patterns

The **Pipes and Filters** pattern is a widely used architectural pattern in software design, particularly for enterprise integration. It involves breaking down complex processes into smaller, reusable steps (filters) that are connected by channels (pipes) through which data flows.

Key Components:

1. **Filters**: Each filter is an independent processing unit that performs a specific task, such as filtering, transforming, or enriching the data. Filters can work in isolation and are typically reusable.
2. **Pipes**: Pipes are the connections between filters. They pass the output of one filter to the next, forming a pipeline. Pipes ensure that each filter can focus solely on its task without needing to know the source or destination of the data.

How It Works:

- Data enters the pipeline through the first filter, which performs its processing and sends the result to the next filter through a pipe.
- This continues through the chain until the final output is generated.
- The system allows for **modularity** and **reusability** of each filter, making the pipeline flexible and easy to extend or modify.

Common Use Cases:

- **Data Processing Pipelines**: Transforming, filtering, or enriching data in steps (e.g., ETL processes).
- **Message Routing**: In an enterprise system, a message might pass through filters that validate, transform, and log the message before it reaches its destination.
- **Microservices**: The pattern can be applied where services communicate asynchronously, using a pipeline for processing the requests.

Benefits:

- **Modularity**: Each filter is independent and can be reused or replaced.
- **Flexibility**: You can easily add or remove filters from the pipeline.
- **Scalability**: The pattern can be scaled by running filters in parallel or in different environments.

In summary, **Pipes and Filters** is a robust pattern for breaking down complex tasks into smaller, manageable steps that work together in a sequence, allowing systems to be more flexible and maintainable. Below is an example of some code that implements it (We used ChatGPT to generate this code).

```
using System;
using System.Threading.Tasks;
using System.Threading.Tasks.Dataflow;

class Program
{
    static void Main()
    {
        // Create the filter block (validates messages)
        var filterBlock = new TransformBlock<string, string>(message =>
        {
            if (message.StartsWith("Valid"))
                return message;  // Passes valid messages
            return null;         // Filters out invalid messages
        });

        // Create the transformation block (converts to uppercase)
        var transformBlock = new TransformBlock<string, string>(message =>
        {
            return message?.ToUpper();  // Transforms valid messages to
uppercase
        });

        // Create the action block (final handler, prints message)
        var actionBlock = new ActionBlock<string>(message =>
        {
            if (message != null)
                Console.WriteLine($"Processed message: {message}");
        });

        // Link the blocks together to form the pipeline
        filterBlock.LinkTo(transformBlock, new DataflowLinkOptions {
PropagateCompletion = true });
```

```
        transformBlock.LinkTo(actionBlock, new DataflowLinkOptions {
PropagateCompletion = true });

        // Post some messages into the pipeline
        filterBlock.Post("Valid message 1");
        filterBlock.Post("Invalid message");
        filterBlock.Post("Valid message 2");

        // Signal completion to the pipeline and wait for processing to finish
        filterBlock.Complete();
        actionBlock.Completion.Wait();

        Console.WriteLine("Processing complete.");
    }
}
```