Migrating StudentERP to a Microservices Architecture with .NET Core

Tilakbhai Hemantkumar Gandhi
Independent Researcher

May 2025

Abstract

This white paper outlines a detailed strategy to transition the StudentERP monolithic web application, developed using ASP.NET Core MVC, into a microservices architecture with .NET Core, aiming to enhance scalability and maintainability for 5,000 users. The goal is to decompose the application into standalone services for user management, student records, and department administration, thereby overcoming the constraints of the existing monolithic structure. The approach involves analyzing the current system, developing a prototype StudentService Web API tailored for 1,000 daily student updates, and assessing deployment and communication methods. Key challenges, including inter-service communication, deployment complexity, logging, and database decentralization, are addressed with solutions such as REST APIs, Kubernetes, Application Insights, and event-driven mechanisms. Moreover, industry examples from Netflix, Amazon, and Uber highlight the advantages of microservices. The paper, published as an independent research effort on Zenodo, concludes that a gradual migration provides substantial scalability benefits for StudentERP, offering a practical roadmap for developers new to microservices.

1 Introduction

The StudentERP application, a monolithic web system developed with ASP.NET Core MVC, supports educational operations for 5,000 users while managing 1,000 student record updates daily. Although monolithic designs are initially simple to build, they encounter scalability and maintenance challenges as user demand grows (1). A microservices approach addresses these issues by enabling autonomous creation, rollout, and scaling of application components (2). Consequently, this paper proposes a migration plan to transform StudentERP into a microservices-based system using .NET Core, drawing on the author's experience with repository patterns and Entity Framework Core. The study encompasses a prototype implementation, industry comparisons, and solutions to technical hurdles, aiming to provide a practical guide for developers. This work was undertaken as an independent research effort and published on Zenodo to share insights with the broader developer community.

2 Architecture Overview

2.1 Current Monolithic Architecture

StudentERP operates as an ASP.NET Core MVC application, employing a database-first methodology with Entity Framework Core and MS SQL Server (4). It comprises three main components: User Management, which oversees authentication and user profiles for 5,000 users; Student Records, managing CRUD operations for student data with 1,000 daily updates; and Department Administration, handling department-related data across multiple faculties. However, all components reside within a single codebase and database, leading to tight coupling and scalability constraints (3).

[Diagram Placeholder: Monolithic Architecture of StudentERP, showing a single ASP.NET Core MVC application connected to a shared MS SQL database]

Figure 1: Monolithic Architecture of StudentERP

2.2 Proposed Microservices Architecture

The proposed design splits StudentERP into three standalone services, each supported by an independent database. UserService manages authentication and user data for 5,000 users, ensuring secure access during peak login times. StudentService oversees student records, handling 1,000 daily updates efficiently. DepartmentService administers department data across faculties. Developers build each service as an ASP.NET Core Web API, which interacts through REST APIs to ensure straightforward communication (4). Furthermore, an API gateway (e.g., Ocelot) directs requests and centralizes authentication for streamlined access.

[Diagram Placeholder: Proposed Microservices Architecture for StudentERP, showing independent services (UserService, StudentService, DepartmentService), each with its own database, connected via an API gateway using REST APIs]

Figure 2: Proposed Microservices Architecture for StudentERP

3 Comparison of Architectures

Table 1 contrasts monolithic and microservices architectures, emphasizing their impact on StudentERP's performance and development process.

4 Real-world Applications

Several organizations have embraced microservices, offering valuable lessons for StudentERP's migration. For instance, Netflix shifted to microservices, cutting latency by 30% and achieving 99.99% uptime through tools like Eureka and

Criterion	Monolithic (StudentERP)	Microservices (Proposed)
Scalability	Scales the entire application, which increases costs unnecessarily.	Scales individual services, optimizing resource usage (1).
Maintenance	Risks unintended side effects during updates due to tight coupling.	Simplifies updates and testing through isolated services (2).
Deployment	Involves a single deployment, which is simpler but riskier for large systems.	Requires multiple deployments, which are complex but fault-tolerant (4).
Fault Tolerance	Suffers from single failures that impact all functionality across the system.	Limits the impact of failures to specific services through isolation (3).

Table 1: Comparison of Monolithic and Microservices Architectures

Cassandra (5), aligning with StudentERP's need for responsive student data access. Similarly, Amazon implemented microservices for AWS, shortening deployment durations to minutes with Lambda and enhancing fault tolerance (6), which could enable faster department data updates in StudentERP. Additionally, Uber adopted microservices to support 10 million daily rides, reducing latency by 25% with a Node.js-based system (7), suggesting potential improvements for StudentERP's student record updates.

5 Prototype Implementation

A prototype StudentService, developed as an ASP.NET Core Web API, reuses the repository pattern from StudentERP while adhering to standard practices (4). It includes a Student model, an IStudentRepository interface, and CRUD endpoints, all connected to a dedicated MS SQL database to handle 1,000 daily updates efficiently.

```
using Microsoft.AspNetCore.Mvc;
using StudentService.Models;
using StudentService.Repositories;

namespace StudentService.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    public class StudentsController : ControllerBase
    {
        private readonly IStudentRepository _repository;

        public StudentsController(IStudentRepository repository)
```

```
        {
            _repository = repository;
        }

        [HttpGet]
        public async Task<IActionResult> GetAll()
        {
            var students = await _repository.GetAllAsync();
            return Ok(students);
        }

        [HttpPost]
        public async Task<IActionResult> Create([FromBody] Student
           student)
        {
            if (student == null) return BadRequest();
            await _repository.AddAsync(student);
            return CreatedAtAction(nameof(GetAll), new { id =
               student.Id }, student);
        }
    }
}
```

```
namespace StudentService.Models
{
    public class Student
    {
        public int Id { get; set; }
        public string Name { get; set; }
        public int DepartmentId { get; set; }
    }
}

namespace StudentService.Repositories
{
    public interface IStudentRepository
    {
        Task<IEnumerable<Student>> GetAllAsync();
        Task AddAsync(Student student);
    }
}
```

6 Challenges and Solutions

The migration presents several challenges, which the following solutions address effectively. First, inter-service communication relies on REST APIs to ensure straightforward interactions; however, gRPC may reduce response times for high-frequency interactions (8), and in StudentERP, StudentService validates tokens by querying UserService via HttpClient for 5,000 users. Second, deploy-

ment complexity is mitigated as Docker containers and Kubernetes streamline the process, as demonstrated in the StudentService Dockerfile (9). Third, Application Insights tracks performance metrics to maintain responsiveness across services (10). Finally, database decentralization risks inconsistencies in student-department relationships, but RabbitMQ ensures eventual consistency with events like StudentUpdated (11).

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["StudentService.csproj", "."]
RUN dotnet restore "StudentService.csproj"
COPY . .
WORKDIR "/src/."
RUN dotnet build "StudentService.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "StudentService.csproj" -c Release -o
   /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "StudentService.dll"]
```

7 Conclusion

This white paper demonstrates that migrating StudentERP to a microservices architecture with .NET Core significantly enhances scalability, maintainability, and fault tolerance for its 5,000 users. Consequently, the StudentService prototype confirms the viability of independent services, leveraging the author's expertise. Industry examples from Netflix, Amazon, and Uber underscore the benefits of microservices. Moreover, challenges like inter-service communication and database decentralization are manageable with REST, Kubernetes, and RabbitMQ. A phased migration, beginning with StudentService by Q3 2025 and deploying to Azure AKS, is recommended. Future work includes developing UserService and integrating Redis for caching, positioning StudentERP for sustained growth.

References

[1] S. Newman, *Building Microservices: Designing Fine-Grained Systems*. O'Reilly Media, 2015.

[2] M. Fowler and J. Lewis, "Microservices: A definition of this new architectural

term," 2014. [Online]. Available: https://martinfowler.com/articles/microservices.html

[3] N. Dragoni et al., "Microservices: Yesterday, Today, and Tomorrow," in *Present and Ulterior Software Engineering*, Springer, 2017, pp. 195–216.

[4] Microsoft, "Microservices architecture style," 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices

[5] Netflix Technology Blog, "Ready for changes with Hexagonal Architecture," 2017. [Online]. Available: https://netflixtechblog.com/ready-for-changes-with-hexagonal-architecture-b315ec967749

[6] Amazon Web Services, "Microservices on AWS," 2016. [Online]. Available: https://aws.amazon.com/microservices/

[7] Uber Engineering, "Building Uber's New Architecture," 2018. [Online]. Available: https://eng.uber.com/microservice-architecture/

[8] gRPC, "Introduction to gRPC," 2023. [Online]. Available: https://grpc.io/docs/what-is-grpc/introduction/

[9] Kubernetes, "Kubernetes Documentation," 2023. [Online]. Available: https://kubernetes.io/docs/home/

[10] Microsoft, "Application Insights Overview," 2023. [Online]. Available: https://learn.microsoft.com/en-us/azure/azure-monitor/app/app-insights-overview

[11] RabbitMQ, "RabbitMQ Tutorials," 2023. [Online]. Available: https://www.rabbitmq.com/getstarted.html