INFORMATICS INSTITUTE OF TECHNOLOGY

In Collaboration with

UNIVERSITY OF WESTMINSTER

Enterprise Application Development 7SENG001C
Coursework 2 by

Mr. K.K. Gayan Sanjeewa

(IIT Student No: 20232961)

(The University of Westminster No: W2084859)

Supervised by

Mr. Nadeesh Herath

Submitted in partial fulfillment of the requirements for the MSc in Advanced Software Engineering degree at the University of Westminster.

July 2024

Contents

1.1. Critical review on why two candidate functionalities were chosen to scale backed by research

In the context of developing a University Student Performance Tracking and Prediction Tool, the initial approach involved creating a monolithic application. While this monolithic architecture is straightforward to implement initially, it can present several challenges as the application grows in complexity and usage. To address these challenges, a decision was made to decouple the CRUD (Create, Read, Update, Delete) operations from the report generation functionality. This separation brings numerous advantages, particularly in terms of scalability, maintainability, and development efficiency. Below, we explore these benefits in detail.

1. Improved Scalability

1.1 Independent Scaling of Services: One of the primary advantages of decoupling CRUD operations from report generation is the ability to scale each service independently. In a monolithic architecture, the entire application must be scaled together, which can be inefficient and costly. By separating these functionalities into distinct services, we can allocate resources based on the specific needs of each service. For instance, CRUD operations might require more frequent access to the database, while report generation might demand higher computational power during specific times, such as the end of a semester. Independent scaling ensures that each service can handle its load efficiently without over-provisioning resources for the entire application.

1.2 Optimized Resource Utilization: Decoupling allows for optimized resource utilization. Since CRUD operations and report generation have different usage patterns, they can be deployed on different types of infrastructure tailored to their specific needs. For example, CRUD operations might benefit from a high-availability database cluster, while report generation could leverage a compute-optimized environment. This targeted allocation of resources leads to better performance and cost savings.

2. Enhanced Maintainability

2.1 Simplified Codebase: Separating CRUD operations and report generation results in a more modular codebase. Each service can be developed, tested, and maintained independently, reducing the complexity of the overall system. This modularity makes it easier for developers to understand and work on specific parts of the application without being overwhelmed by the entire codebase.

2.2 Easier Debugging and Testing: With decoupled services, debugging and testing become more manageable. Issues can be isolated to specific services, making it easier to identify and fix bugs. Automated tests can be run independently for each service, ensuring that changes in one service do not inadvertently affect others. This isolation improves the reliability and stability of the application.

3. Development Efficiency

3.1 Parallel Development: Decoupling enables parallel development, where different teams can work on CRUD operations and report generation simultaneously. This parallelism accelerates the development process, allowing for faster feature releases and updates. Teams can focus on their specific areas of expertise, leading to higher quality code and more innovative solutions.

3.2 Flexibility in Technology Stack: When services are decoupled, each can be developed using the most appropriate technology stack for its requirements. For example, CRUD operations might be implemented using a relational database and a RESTful API, while report generation could leverage big data tools and machine learning frameworks. This flexibility allows for the use of best-of-breed technologies, enhancing the overall functionality and performance of the application.

4. Event-Driven Architecture

4.1 Efficient Event Handling: Moving to an event-driven architecture further enhances the benefits of decoupling. In this model, services communicate through events, which are triggered by specific actions or changes in the system. For instance, when a new study session is recorded, an event can be triggered to update the user's progress and generate a report. This approach ensures that only the relevant services are activated in response to an event, leading to more efficient processing and reduced latency.

4.2 Asynchronous Processing: Event-driven architectures support asynchronous processing, where tasks are executed independently and do not block the main application flow. This is particularly useful for report generation, which can be a resource-intensive operation. By processing reports asynchronously, the system can continue to handle CRUD operations without delay, improving the overall user experience.

5. Feature Release and Scaling

5.1 Incremental Feature Releases: Decoupling services allows for incremental feature releases. New features can be added to specific services without affecting the entire application. For example, enhancements to the report generation algorithm can be deployed independently of CRUD operations. This incremental approach reduces the risk of introducing bugs and allows for more frequent updates.

5.2 Scalable Feature Rollout: As the application grows, certain features may require more resources or specialized infrastructure. Decoupling allows for scalable feature rollout, where new features can be deployed on dedicated infrastructure tailored to their needs. This ensures

that the application can handle increased load and complexity without compromising performance.

6. Zero Downtime Deployment

6.1 Seamless Updates: Decoupling services facilitates zero downtime deployment. Updates to one service can be deployed without taking the entire application offline. This is particularly important for applications that require high availability. For instance, updates to the report generation service can be rolled out while CRUD operations continue to function normally, ensuring uninterrupted access for users.

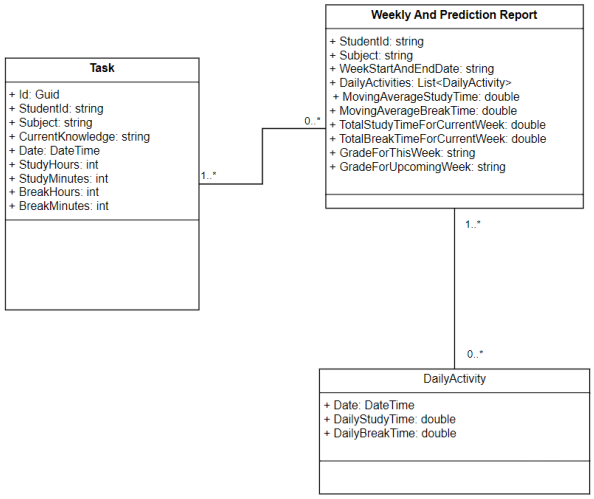
6.2 Reduced Deployment Risk: By isolating services, the risk associated with deployments is significantly reduced. If an update to one service fails, it does not impact the other services. This isolation allows for more frequent and safer deployments, as issues can be quickly identified and resolved without affecting the entire application.

7. Future-Proofing and Extensibility

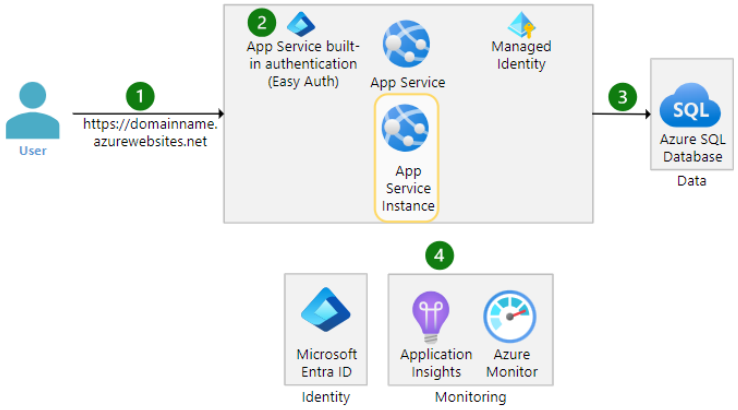
7.1 Exposing Report Generation as a Service: One of the significant advantages of decoupling the report generation service is the potential to expose it as a standalone service to third-party applications. This can be achieved with minimal effort since the service is already isolated and can be accessed via well-defined APIs. This extensibility opens up new opportunities for integration with other systems, such as learning management systems (LMS) or external analytics platforms, providing additional value to users.

7.2 Modular Expansion: Decoupling allows for modular expansion of the application. As new requirements emerge, additional services can be developed and integrated without disrupting existing functionality. For example, a new predictive analytics service could be added to enhance the tool's ability to forecast academic performance based on study patterns. This modularity ensures that the application can evolve and adapt to changing user needs and technological advancements.

1.2. Class Diagram for the architecture



1.3 Deployment Diagram for the architecture



- **Microsoft Entra ID** : cloud-based identity and access management service. which provides a single identity control plane to manage permissions and roles for users accessing your web application. It integrates with App Service.
- **App Service** is a fully managed platform for building, deploying, and scaling web applications.
- **Azure Monitor** is a monitoring service that collects, analyzes, and acts on telemetry data across your deployment.
- **Azure SQL Database** is a managed relational database service for relational data.

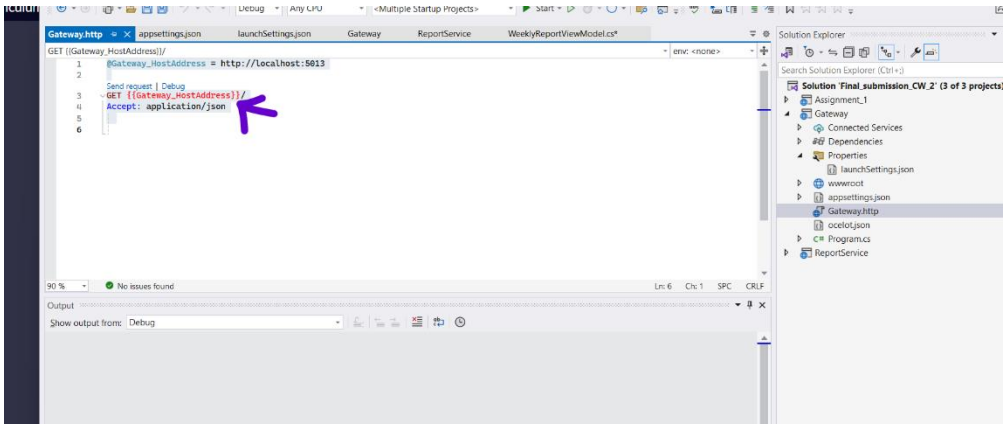
1. A Student issues an HTTPS request to the App Service's default domain on azurewebsites.net in above diagram , for this app it's possible to buy particular domain. This domain automatically points to Student Progress Track App Service's built-in public IP. The TLS connection is established from the client directly to app service. The certificate is managed completely by Azure.
2. Easy Auth : feature of Azure App Service, ensures that the Student accessing the site is authenticated with Microsoft Entra ID.
3. Your application code deployed to App Service handles the request. For example, that code might connect to an Azure SQL Database instance, using a connection string configured in the App Service configured as an app setting.
4. The information about original request to App Service and the call to Azure SQL Database are logged in Application Insights.

2.1 CRUD operations via Web Services

CRUD operations (Create, Read, Update, Delete) are fundamental to web services, enabling the manipulation of data in a structured and efficient manner. In the provided StudentActivityController class, these operations are implemented using ASP.NET Core MVC, showcasing how web services can manage data interactions with a database. The Add method facilitates the creation of new student activity records by accepting input through a form and saving it to the database using Entity Framework Core's AddAsync and SaveChangesAsync methods. The List method retrieves all records from the StudentActivityTrackRecord table, displaying them to the user. This is achieved through the asynchronous ToListAsync method, ensuring non-blocking data retrieval.

For updating records, the Edit method is employed. It first fetches the specific record by its Id using FindAsync, then updates the relevant fields, and finally saves the changes back to the database. This ensures that the data remains consistent and up-to-date. The Delete method removes a record from the database, identified by its Id, using the Remove method followed by SaveChangesAsync to persist the deletion.

These CRUD operations are essential for maintaining the integrity and accessibility of data within web applications. By leveraging asynchronous methods, the controller ensures that the application remains responsive, even during intensive data operations. Additionally, the use of Entity Framework Core simplifies the interaction with the database, providing a robust and scalable solution for data management. This approach not only enhances the performance and reliability of the web service but also ensures a seamless user experience by efficiently handling data operations in the background via REST Protocol by accepting JSON values.



2.2. Effective use of a repository options

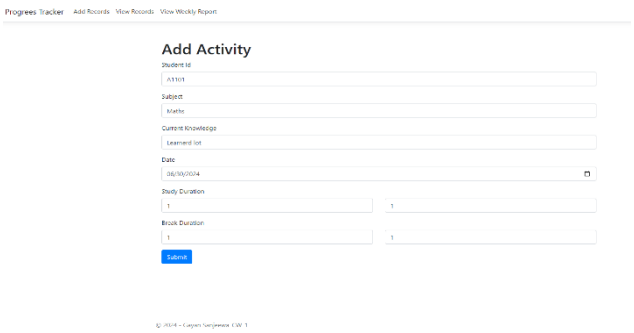
In modern software development, the effective use of repository options is crucial for optimizing data access and ensuring the scalability and maintainability of applications. A repository pattern abstracts the data access layer, providing a clean separation between the business logic and data access code. This abstraction not only enhances code readability and testability but also allows for more flexible and efficient data management strategies.

One of the primary benefits of using a repository pattern is the ability to centralize data access logic. By encapsulating the data access code within repositories, developers can ensure that all data interactions are consistent and adhere to the same set of rules and validations. This centralization simplifies maintenance, as any changes to the data access logic need to be made in only one place, reducing the risk of bugs and inconsistencies.

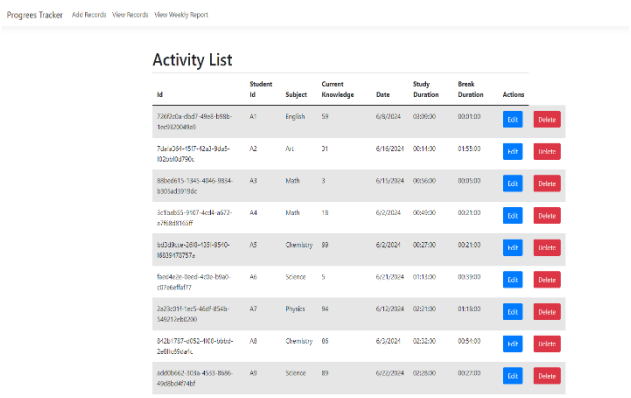
Repositories also facilitate the implementation of caching strategies. By introducing a caching layer within the repository, frequently accessed data can be stored in memory, significantly reducing the number of database queries and improving application performance. This is particularly beneficial in read-heavy applications where the same data is requested multiple times. Implementing caching within the repository ensures that the caching logic is transparent to the rest of the application, maintaining a clean separation of concerns (Game, 2019). As a future improvement for report service, it's possible to add this.

The repository pattern also supports the implementation of advanced querying capabilities. By leveraging LINQ (Language Integrated Query) in repositories, developers can construct complex queries in a type-safe manner, reducing the likelihood of runtime errors.

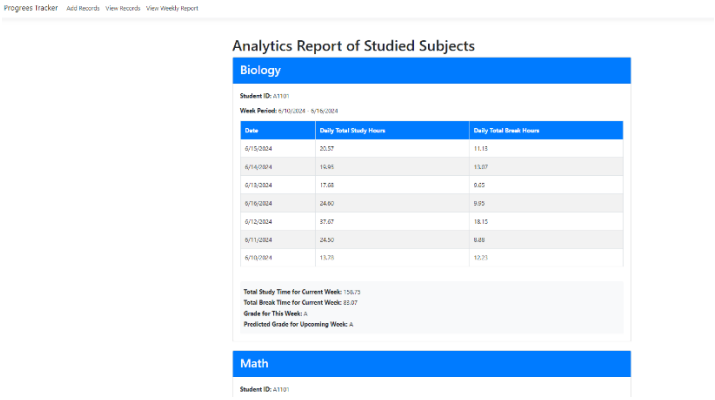
Furthermore, repositories can be designed to support multiple data sources. In applications that need to interact with different databases or external services, the repository pattern provides a unified interface for data access. This abstraction allows the application to switch between different data sources without significant changes to the business logic, enhancing the application's flexibility and adaptability.



Add Activity



View Activity List



Weekly Analytic Report

2.3. Usage of Threads for I/O operations to improve usability

- There were use case needed preload some data , in that case to improve the performance used the async calls for those.

```
0 references
public StudentActivityController(ApplicationDbContext dbContext)
{
    this.dbContext = dbContext;
    Task.Run(() => GenerateSampleDataAsync()).Wait(); // Run data generation asynchronously
}
```

Async calls

- Since application requests data from database, used the thread safe mechanism to avoid such issue occurred in multithread environment.

```
await semaphoreSlim.WaitAsync(); // Ensure thread safety when accessing the database context
try
{
    dbContext.StudentActivityTrackRecord.AddRange(activityRecords);
    await dbContext.SaveChangesAsync();
}
finally
{
    semaphoreSlim.Release();
}
```

Thread Safety

- Without repeating the same codes , used reusable functions . Some can be used from anywhere based on the usage

```
1 reference
private static int GetIso8601WeekOfYear(DateTime time)
{
    DayOfWeek day = CultureInfo.InvariantCulture.Calendar.GetDayOfWeek(time);
    if (day >= DayOfWeek.Monday && day <= DayOfWeek.Wednesday)
    {
        time = time.AddDays(3);
    }
    return CultureInfo.InvariantCulture.Calendar.GetWeekOfYear(time, CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
}

2 references
private DateTime GetWeekStartDate(int weekOfYear)
{
    var jan1 = new DateTime(DateTime.Now.Year, 1, 1);
    var daysOffset = DayOfWeek.Monday - jan1.DayOfWeek;
    var firstMonday = jan1.AddDays(daysOffset);
    var firstWeek = CultureInfo.CurrentCulture.Calendar.GetWeekOfYear(jan1, CalendarWeekRule.FirstFourDayWeek, DayOfWeek.Monday);
    var weekNum = weekOfYear;
    if (firstWeek <= 1)
    {
        weekNum -= 1;
    }
    return firstMonday.AddDays(weekNum * 7);
}
```

Reusability

2.4. Usage of advanced techniques

- Consider the thread safety by adding locking mechanism form semaphore.

```
await _semaphoreSlim.WaitAsync(); // Ensure thread safety when accessing the database context
try
{
    await _dbContext.StudentActivityTrackRecord.AddAsync(activityRecord);
    await _dbContext.SaveChangesAsync();
}
finally
{
    _semaphoreSlim.Release();
}
```

Adding Locking mechanism

- Database Grouping and Aggregation: The GroupBy and Select methods are used to perform the grouping and aggregation directly in the database. This reduces the amount of data transferred to the application and leverages the database's optimized query execution.
- Asynchronous Query Execution: In read scenarios , ToListAsync method is used to execute the query asynchronously, improving the responsiveness of the application.
- Fetch Data First: The ToListAsync method is used to fetch all records from the database first. And also for further optimize, you can filter the data by a date range before fetching it from the database

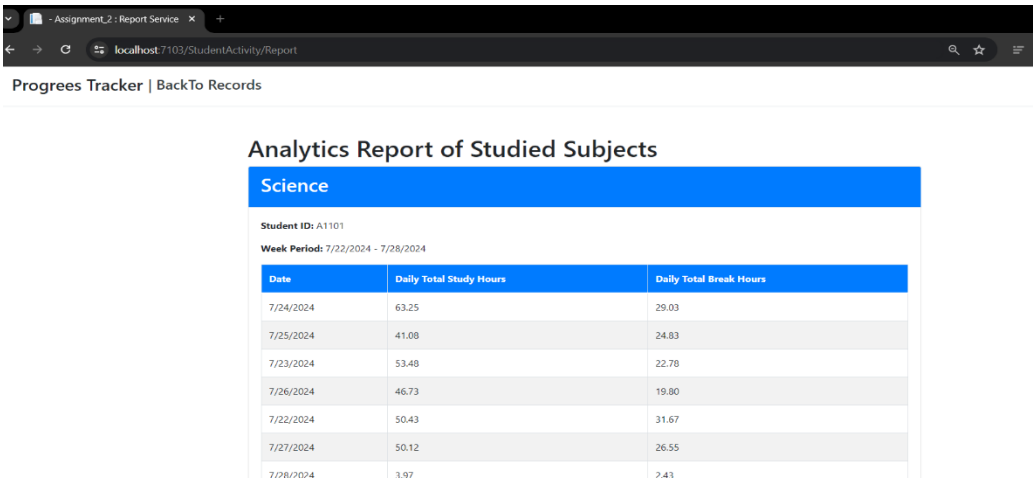
```
[HttpGet]
0 references
public async Task<IActionResult> Report()
{
    var startDate = DateTime.Now.AddMonths(-1); // Example: last month
    var endDate = DateTime.Now;

    var activityRecords = await dbContext.StudentActivityTrackRecord
        .Where(r => r.Date >= startDate && r.Date <= endDate)
        .ToListAsync();
```

Async database calls

- Used the API gateway using Ocelot package and routed the call using ocelot.json file.

```
{
  "Routes": [
    {
      "DownstreamPathTemplate": "/StudentActivity/Add",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "7214"
        }
      ],
      "UpstreamPathTemplate": "/records/add",
      "UpstreamHttpMethod": [ "GET", "POST" ]
    },
    {
      "DownstreamPathTemplate": "/StudentActivity/Add",
      "DownstreamScheme": "https",
      "DownstreamHostAndPorts": [
        {
          "Host": "localhost",
          "Port": "7214"
        }
      ],
```

Ocelot json



Redirected from API Gateway

2.5. References

(Game, 2019) (Koyuncu, 2017)