This cheat sheet is for the course Learn C# Full Stack Development with Angular and ASP.NET by Jannick Leismann.

REPOSITORY DESIGN PATTERN

A well-known design pattern in software engineering for separating an application's business logic from its data access logic is the repository design pattern. It separates the data storage, retrieval, and mapping to domain objects from the rest of the program and offers a collection-like interface for interacting with domain objects.

Advantages

Separation of Concerns

This keeps the business logic and data access logic apart, which simplifies and improves the maintainability of the codebase.

Testability

Enables the usage of mock repositories, which simplifies unit testing.

Flexibility

Allows for simpler changes to the ORM framework or data source without compromising business logic.

Repository Interface

Establishes the terms of data access techniques contracts. It usually contains CRUD (Create, Read, Update, Delete) methods, but it can also be customized with more query methods based on the requirements of the application.

Concrete Repository

Implements the repository interface. This class contains the actual code to interact with the data source, such as a database, an API, or an in-memory collection.

Simple Example

IEmployeeRepository.cs

```
using EmployeeManagement.Models;

namespace EmployeeManagement.Repositories
{
    public interface IEmployeeRepository
    {
        Task<IEnumerable<Employee>> GetAllAsync();
        Task<Employee?> GetByIdAsync(int id);
        Task AddEmployeeAsync(Employee employee);
        Task UpdateEmployeeAsync(Employee employee);
        Task DeleteEmployeeAsync(int id);
    }
}
```

```
using EmployeeManagement.Data;
using EmployeeManagement.Models;
using Microsoft.EntityFrameworkCore;
namespace EmployeeManagement.Repositories
{
    public class EmployeeRepository : IEmployeeRepository
    {
        private readonly AppDbContext _context;
        public EmployeeRepository(AppDbContext context)
        {
            _context = context;
        }
        public async Task AddEmployeeAsync(Employee employee)
        {
```

```
            await _context.Employees.AddAsync(employee);
            _context.SaveChanges();
        }
        public async Task DeleteEmployeeAsync(int id)
        {
            var employeeInDb = await _context.Employees.FindAsync(id);

            if (employeeInDb == null)
            {
                throw new KeyNotFoundException($"Employee with id {id} was
not found.");
            }
            _context.Remove(employeeInDb);
            await _context.SaveChangesAsync();
        }
        public async Task<IEnumerable<Employee>> GetAllAsync()
        {
            return await _context.Employees.ToListAsync();
        }
        public async Task<Employee?> GetByIdAsync(int id)
        {
            return await _context.Employees.FindAsync(id);
        }
        public async Task UpdateEmployeeAsync(Employee employee)
        {
            _context.Employees.Update(employee);
            await _context.SaveChangesAsync();
        }
    }
}
```