Серверная часть приложения реализована в виде стандартной луковой архитектуры. Она имеет несколько слоёв.

Центральным слоем является слой доменных сущностей. Этот слой предоставляет набор анемичных сущностей, которые соответствуют сущностям в базе данных. Пример такой сущности представлен на листинге 4.1.

Листинг 4.1 – Team.cs

public partial class Team

{

public Guid? Id { get; set; }

public string? Name { get; set; }

public string? Country { get; set; }

public string Sport { get; set; } = null!;

public virtual ICollection<Event> EventTeam1s { get; set; } = new List<Event>();

public virtual ICollection<Event> EventTeam2s { get; set; } = new List<Event>();

}

Следующим идёт слой доступа к данным. В приложении используется паттерн репозитория. Репозиторий реализует всю необходимую логику для конкретной сущности по работе с базой данных. Пример репозитория показан на листинге 4.2. Первой задачей слоя доступа к данным является получить данные из базы данных и привести их в форму, пригодную для последующей их обработки в коде. Второй задачей слоя доступа к данным является правильное сохранение данных в базе данных.

Листинг 4.2 – TeamRepository.cs

/// <inheritdoc cref="ITeamRepository"/>

public class TeamRepository : ITeamRepository

{

private DbContext DbContext { get; }

private DbSet<Team> Teams { get; }

private readonly IMapper \_mapper;

public TeamRepository(DbContext context, IMapper mapper)

{

DbContext = context;

\_mapper = mapper;

Teams = context.Set<Team>();

}

/// <inheritdoc/>

public async Task AddAsync(TeamDto team, CancellationToken cancellationToken)

{

var teamEntity = \_mapper.Map<Team>(team);

await Teams.AddAsync(teamEntity, cancellationToken);

await DbContext.SaveChangesAsync(cancellationToken);

}

/// <inheritdoc/>

public async Task DeleteAsync(Guid id, CancellationToken cancellationToken)

{

var teamEntity = await Teams.AsNoTracking().FirstOrDefaultAsync(x => x.Id == id, cancellationToken);

if (teamEntity == null) throw new EntityNotFoundException();

Teams.Remove(teamEntity);

await DbContext.SaveChangesAsync(cancellationToken);

}

/// <inheritdoc/>

public async Task<TeamDto> GetByIdAsync(Guid id, CancellationToken cancellationToken)

{

var teamEntity = await Teams.AsNoTracking().FirstOrDefaultAsync(x => x.Id == id, cancellationToken);

if (teamEntity == null) throw new EntityNotFoundException();

return \_mapper.Map<TeamDto>(teamEntity);

}

/// <inheritdoc/>

public async Task<ICollection<TeamDto>> SearchAsync(string searchString, CancellationToken cancellationToken)

{

var teams = await Teams.AsNoTracking()

.Where(x => x.Name.ToLower().Contains(searchString.ToLower()))

.ProjectTo<TeamDto>(\_mapper.ConfigurationProvider)

.ToListAsync(cancellationToken);

return teams;

}

/// <inheritdoc/>

public async Task UpdateAsync(TeamDto team, CancellationToken cancellationToken)

{

var teamEntity = \_mapper.Map<Team>(team);

Teams.Update(teamEntity);

await DbContext.SaveChangesAsync(cancellationToken);

}

}

После слоя доступа к данным идёт слой бизнес-логики. Этот слой отвечает за всю внутреннюю логику приложения, такую как авторизация пользователей, контроль за корректностью данных и управление слоем доступа к данным. Пример модуля из слоя бизнес-логики приведён на листинге 4.3. Этот слой отделён от остальных портами, к которым остальные слои имеют свои адаптеры. Слой доступа к данным соединён со слоем бизнес-логики управляемыми адаптерами, это означает, что слой бизнес-логики управляет слоем доступа к данным как раз через эти адаптеры. Также слой бизнес-логики соединён с презентационным слоем управляющими портами, которые указывают, что должен выполнить слой бизнес-логики.

Листинг 4.3 – TeamService.cs

/// <inheritdoc cref="ITeamService"/>

public class TeamService(ITeamRepository repository, IMapper mapper) : ITeamService

{

/// <inheritdoc/>

public async Task DeleteAsync(Guid id, CancellationToken cancellationToken)

{

await repository.DeleteAsync(id, cancellationToken);

}

/// <inheritdoc/>

public async Task<TeamDto> GetByIdAsync(Guid id, CancellationToken cancellationToken)

{

return await repository.GetByIdAsync(id, cancellationToken);

}

/// <inheritdoc/>

public async Task<ICollection<TeamDto>> SearchAsync(string searchString, CancellationToken cancellationToken)

{

return await repository.SearchAsync(searchString, cancellationToken);

}

/// <inheritdoc/>

public async Task AddAsync(AddTeamRequest request, CancellationToken cancellationToken)

{

var team = mapper.Map<TeamDto>(request);

await repository.AddAsync(team, cancellationToken);

}

/// <inheritdoc/>

public async Task UpdateAsync(Guid id, UpdateTeamRequest request, CancellationToken cancellationToken)

{

var team = await repository.GetByIdAsync(id, cancellationToken);

team.Name = request.Name;

team.Country = request.Country;

await repository.UpdateAsync(team, cancellationToken);

}

}

Презентационный слой предоставляет собой внешний программный интерфейс, через который фронтенд будет подавать команды и данные бэкенду. Пример модуля из презентационного слоя приведён на листинге 4.4.

Листинг 4.4 – TeamController.cs

/// <summary>

/// Команды.

/// </summary>

[ApiController]

[Route("[controller]")]

[ProducesResponseType((int)HttpStatusCode.InternalServerError)]

public class TeamController(ITeamService service) : BaseController

{

/// <summary>

/// Добавляет команду по модели запроса.

/// </summary>

/// <param name="request">Запрос.</param>

/// <param name="token">Токен отмены.</param>

/// <returns></returns>

[Authorize(Roles = "Worker")]

[HttpPost]

[ProducesResponseType((int)HttpStatusCode.Created)]

[ProducesResponseType((int)HttpStatusCode.Forbidden)]

[ProducesResponseType((int)HttpStatusCode.Unauthorized)]

public async Task<IActionResult> AddAsync(AddTeamRequest request, CancellationToken token)

{

await service.AddAsync(request, token);

return Created();

}

/// <summary>

/// Обновляет команду по модели запроса.

/// </summary>

/// <param name="id">Идентификатор команды.</param>

/// <param name="request">Запрос.</param>

/// <param name="token">Токен отмены.</param>

/// <returns></returns>

[Authorize(Roles = "Worker")]

[HttpPut("{id}")]

[ProducesResponseType((int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.Forbidden)]

[ProducesResponseType((int)HttpStatusCode.Unauthorized)]

public async Task<IActionResult> UpdateAsync(Guid id, UpdateTeamRequest request, CancellationToken token)

{

await service.UpdateAsync(id, request, token);

return Ok();

}

/// <summary>

/// Удаляет команду по идентификатору..

/// </summary>

/// <param name="id">Идентификатор команды.</param>

/// <param name="token">Токен отмены.</param>

/// <returns></returns>

[Authorize(Roles = "Worker")]

[HttpDelete("{id}")]

[ProducesResponseType((int)HttpStatusCode.NoContent)]

[ProducesResponseType((int)HttpStatusCode.Forbidden)]

[ProducesResponseType((int)HttpStatusCode.Unauthorized)]

public async Task<IActionResult> DeleteAsync(Guid id, CancellationToken token)

{

await service.DeleteAsync(id, token);

return NoContent();

}

/// <summary>

/// Получает команду по идентификатору.

/// </summary>

/// <param name="id">Идентификатор.</param>

/// <param name="token">Токен отмены.</param>

/// <returns>Модель команды.</returns>

[Authorize]

[HttpGet("{id}")]

[ProducesResponseType(typeof(TeamDto), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.Unauthorized)]

public async Task<IActionResult> GetByIdAsync(Guid id, CancellationToken token)

{

var team = await service.GetByIdAsync(id, token);

return Ok(team);

}

/// <summary>

/// Получает команды по строке поиска.

/// </summary>

/// <param name="searchString">Строка поиска.</param>

/// <param name="token">Токен отмены.</param>

/// <returns>Коллекция моделей команд.</returns>

[Authorize]

[HttpGet("Search/{searchString}")]

[ProducesResponseType(typeof(TeamDto), (int)HttpStatusCode.OK)]

[ProducesResponseType((int)HttpStatusCode.Unauthorized)]

public async Task<IActionResult> SearchAsync(string searchString, CancellationToken token)

{

var teams = await service.SearchAsync(searchString, token);

return Ok(teams);

}

}

Также были реализованы контракты для обмена данными между слоями приложения, а также между фронтендом и бэкендом. Пример контракта показан на листинге 4.5.

Листинг 4.5 – TeamDto.cs

public class TeamDto

{

public Guid? Id { get; set; }

public string? Name { get; set; }

public string? Country { get; set; }

public string Sport { get; set; } = null!;

}

Кроме вышеописанных технологий в реализации приложения применяется автомаппер, который упрощает преобразование доменных сущностей в контрактные.  
 База данных и внешний программный интерфейс поднимаются в виде контейнеров в Docker. Docker упрощает процесс развертывания приложения на различных серверах. Контейнеры можно легко копировать, перемещать и запускать на любых машинах с поддержкой Docker, что ускоряет и упрощает развертывание. Кроме того, Docker-контейнеры поддерживают горизонтальное масштабирование, что позволяет легко увеличить количество экземпляров базы данных.  
 Авторизация в приложении реализована с помощью JWT.  
 В качестве ORM для реализации подключения и работы с базой данных был выбран фреймворк EF Core. EF Core предоставляет уровень абстракции над базой данных, что позволяет работать с различными СУБД (например, SQL Server, PostgreSQL, MySQL) без изменения кода приложения. EF Core поддерживает механизм миграций, который упрощает управление изменениями структуры базы данных. Миграции позволяют автоматически применять изменения в схеме базы данных при обновлении приложения, что облегчает процесс разработки и развертывания. EF Core является частью экосистемы .NET, что обеспечивает плотную интеграцию с другими компонентами и технологиями платформы. Это позволяет использовать весь потенциал .NET для создания мощных и производительных приложений. Благодаря улучшенной производительности и оптимизациям, EF Core позволяет эффективно работать с большими объемами данных и выполнять сложные запросы. Это делает его подходящим выбором для высоконагруженных приложений. EF Core автоматически отслеживает изменения в объектах и управляет состоянием сущностей, что упрощает выполнение операций обновления и удаления. Это снижает вероятность ошибок и облегчает управление данными.  
 С помощью EF из базы данных были получены доменные сущности и контекст базы данных