

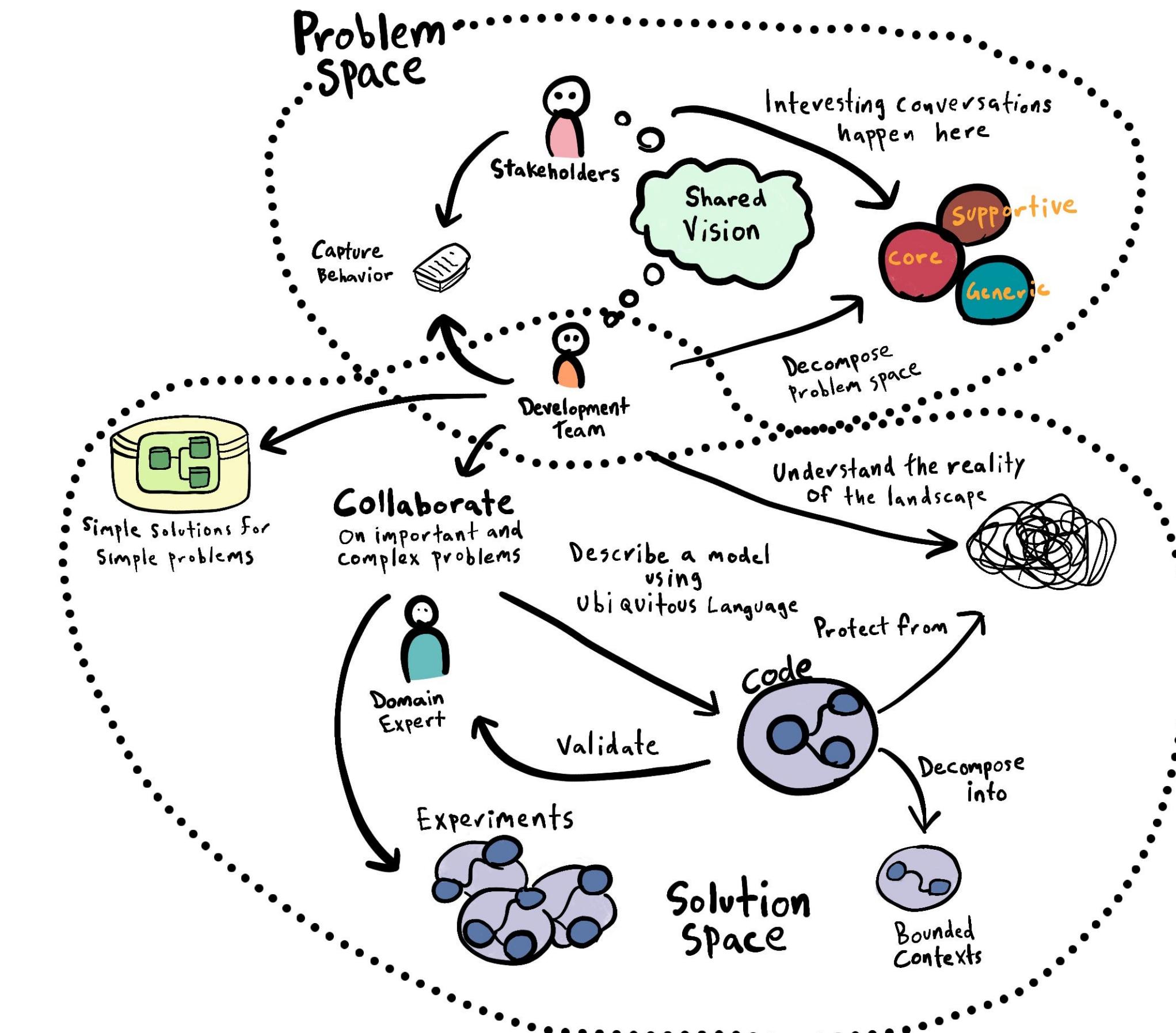
DDD, чистая архитектура и сеть

07.11.2025, ФКН ПИ

Определение

DDD (Domain Driven Design) – подход к проектированию, основанный на модели предметной области.

Предметная область – область человеческой деятельности, определяемая общим набором понятий и отношений, используемых совместно



Основные концепции DDD

Единый язык (Ubiquitous Language) – это общий язык, используемый как разработчиками, так и экспертами в предметной области. Он помогает устранить недопонимание между техническими и бизнес-специалистами.

Пример:

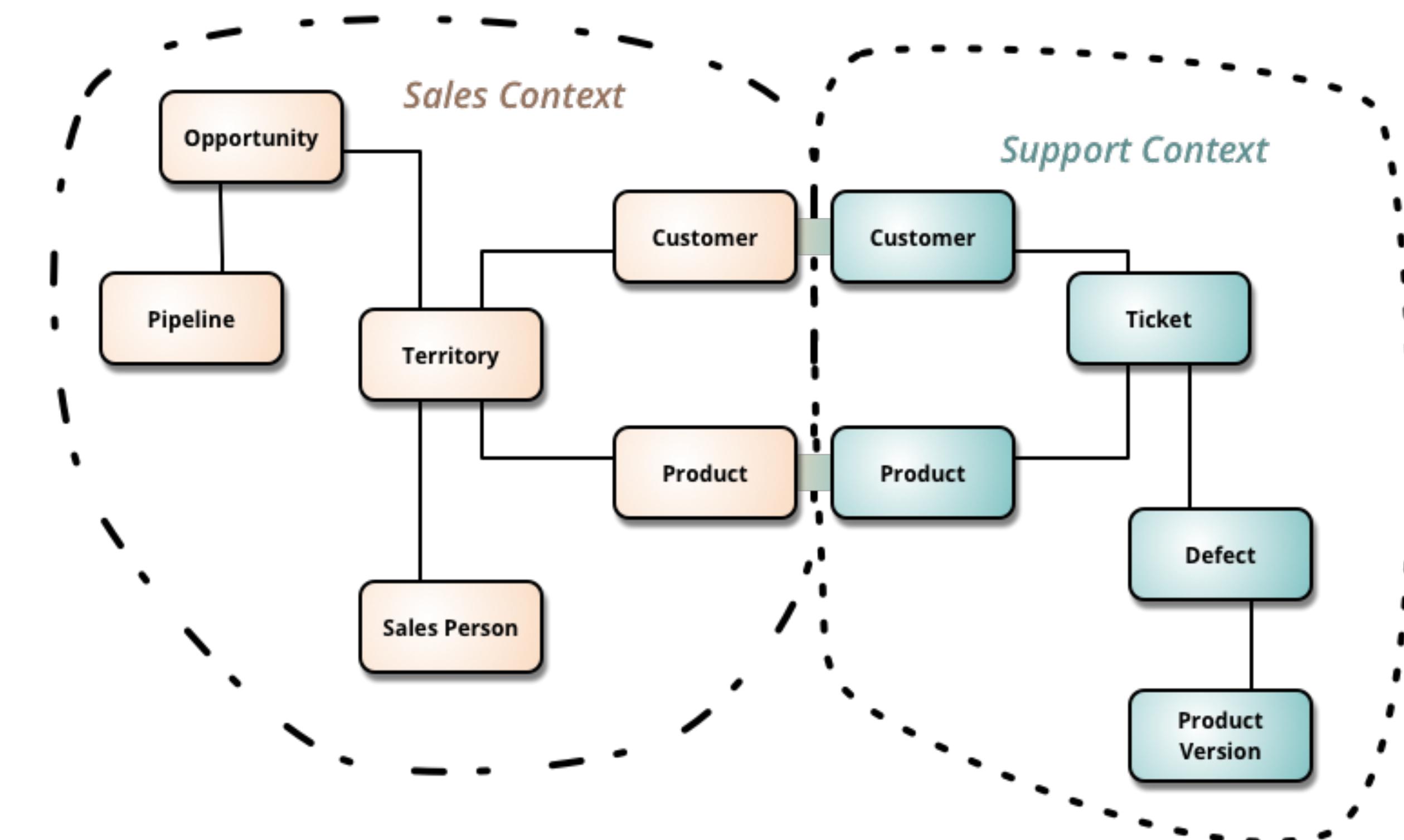
Вместо использования технических терминов вроде “Entity” или “Repository”, мы используем термины из бизнес-домена, такие как “Покупатель”, “Автомобиль”, “Продажа”.

Основные концепции DDD

Ограниченный контекст (Bounded Context) – это граница, внутри которой определенная модель имеет конкретное значение. Разные контексты могут иметь разные модели для одних и тех же понятий.

Пример:

Понятие “Клиент” в контексте продаж может отличаться от понятия “Клиент” в контексте бухгалтерии.



Строительные блоки DDD. Value-objects

Объекты-значения (Value Objects) – это объекты, которые не имеют идентичности и полностью определяются своими атрибутами.



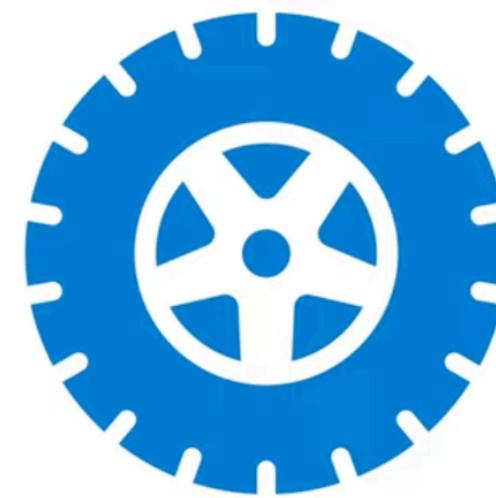
Red: 176
Green: 83
Blue: 167



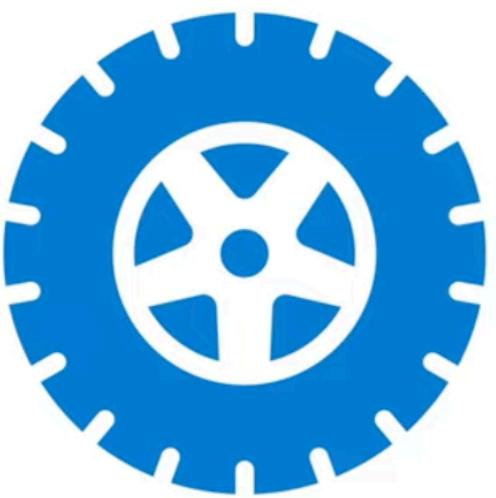
Red: 176
Green: 83
Blue: 167

Строительные блоки DDD. Сущности

Сущности (Entities) – это объекты, которые имеют идентичность, не зависящую от их атрибутов. Две сущности могут быть равны, даже если их атрибуты различаются.



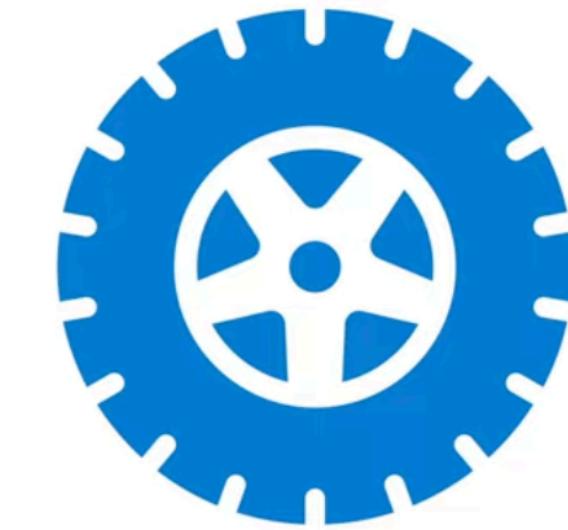
Size: 450mm
Type: Snow
Air: 29psi



Size: 450mm
Type: Snow
Air: 29psi

Строительные блоки DDD. Domain Events

События предметной области (Domain Events) – это объекты, которые фиксируют что-то, что произошло в предметной области.



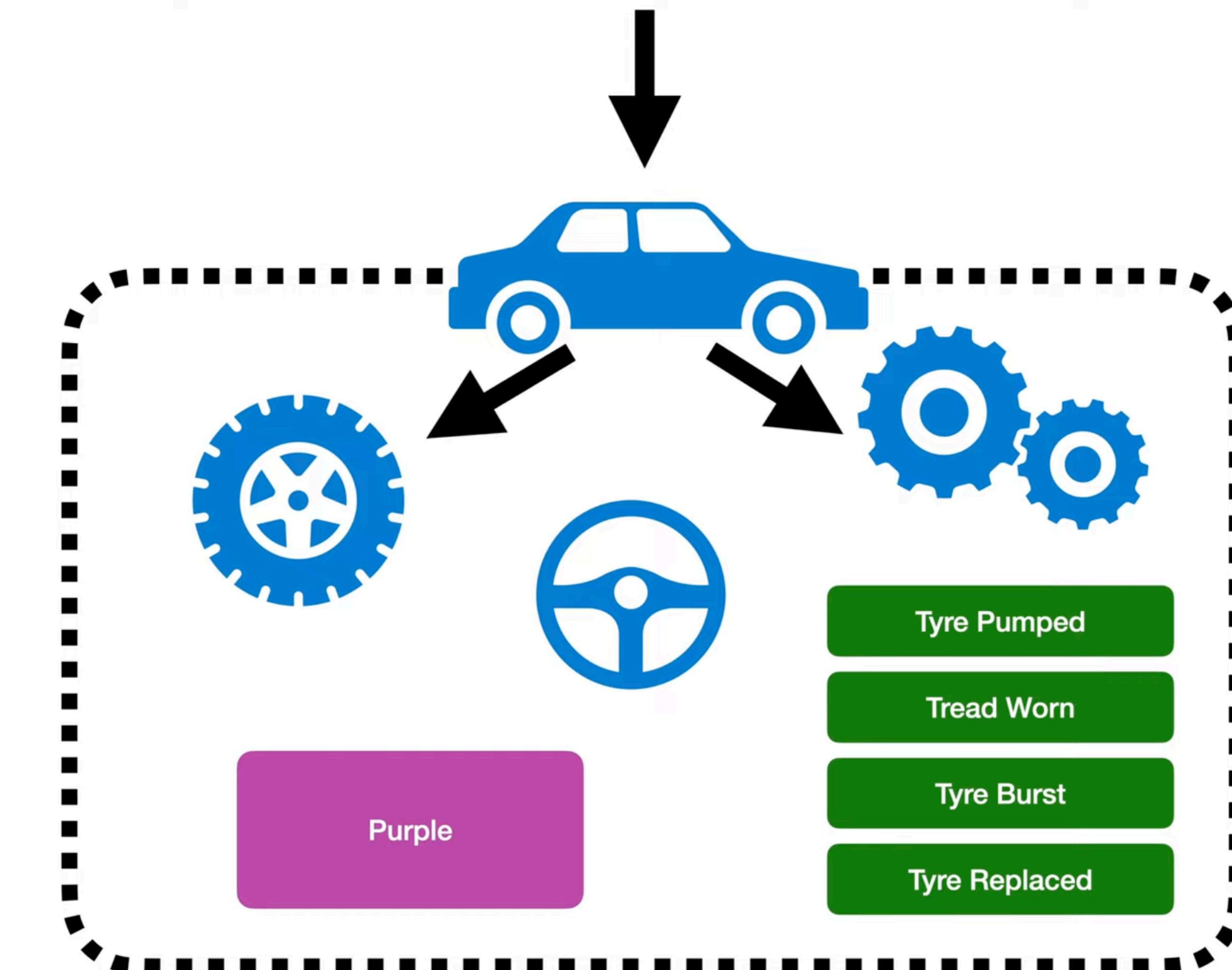
Tyre Pumped

Size: 450mm
Type: Snow
Air: 35psi
Id: 1234

Id: 1234
OldAir: 29psi
NewAir: 35psi

Строительные блоки DDD. Агрегаты

Агрегаты (Aggregates) – это кластер объектов, которые рассматриваются как единое целое с точки зрения изменения данных. У каждого агрегата есть корень (Aggregate Root), через который осуществляется доступ к другим объектам агрегата.



Строительные блоки DDD. Репозитории

Репозитории (*Repositories*) – это механизм для инкапсуляции хранения, поиска и извлечения объектов модели.

Для чего?

1. Инкапсуляция доступа к данным

Репозиторий скрывает детали работы с базой данных (SQL-запросы, ORM, кеширование) и предоставляет удобные методы для работы с сущностями.

2. Отделение доменной логики от инфраструктуры

Позволяет доменному слою не зависеть от конкретной технологии хранения данных (EF Core, Dapper, MongoDB и т. д.).

3. Обеспечение работы с агрегатами

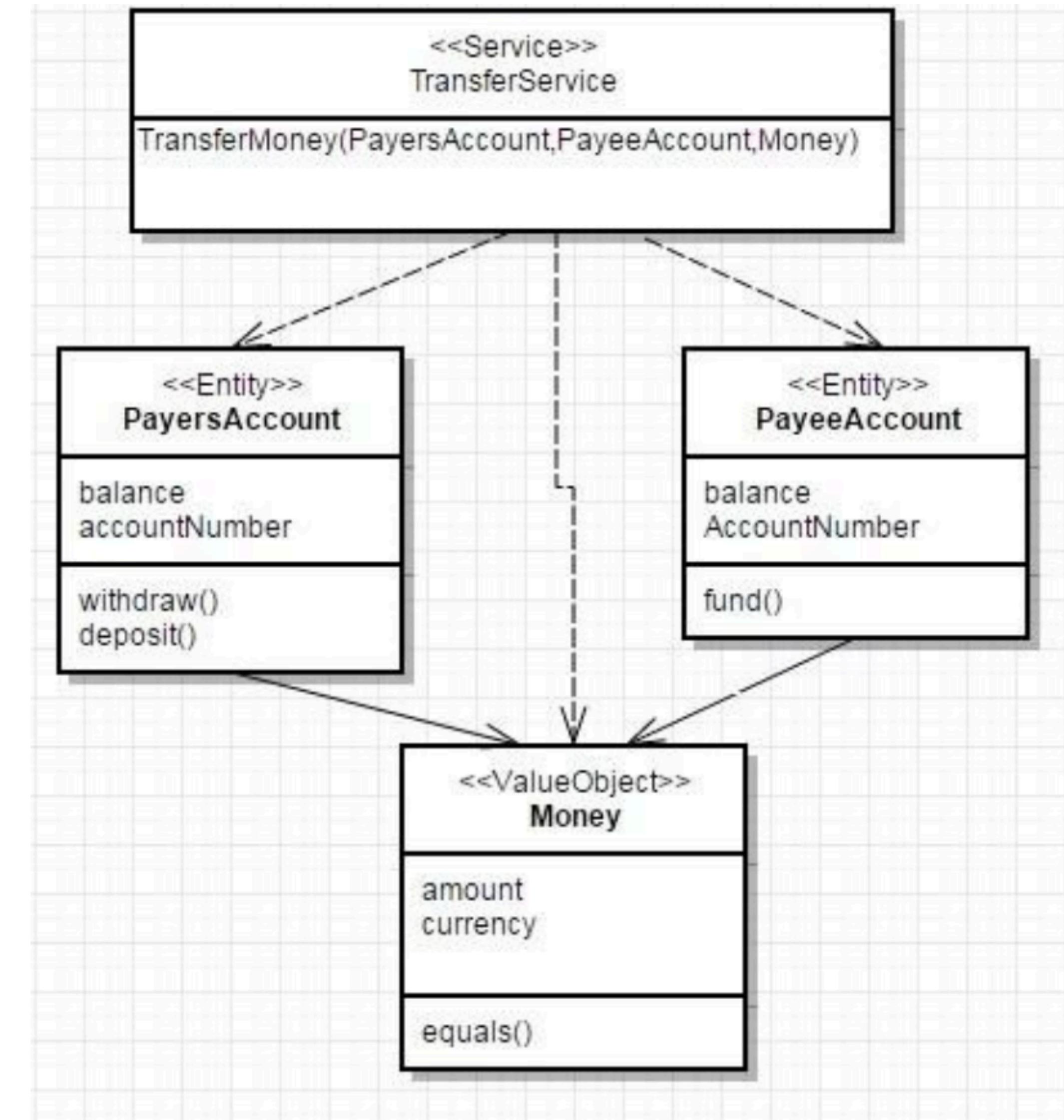
В DDD есть понятие **агрегатов (Aggregates)** — групп объектов, которые изменяются и сохраняются как единое целое. Репозиторий управляет агрегатами, гарантируя целостность данных.

4. Упрощение тестирования

Благодаря репозиториям можно легко подменять реальное хранилище мок-объектами (например, `InMemoryRepository`), что делает тесты независимыми от БД.

Строительные блоки DDD. Сервисы

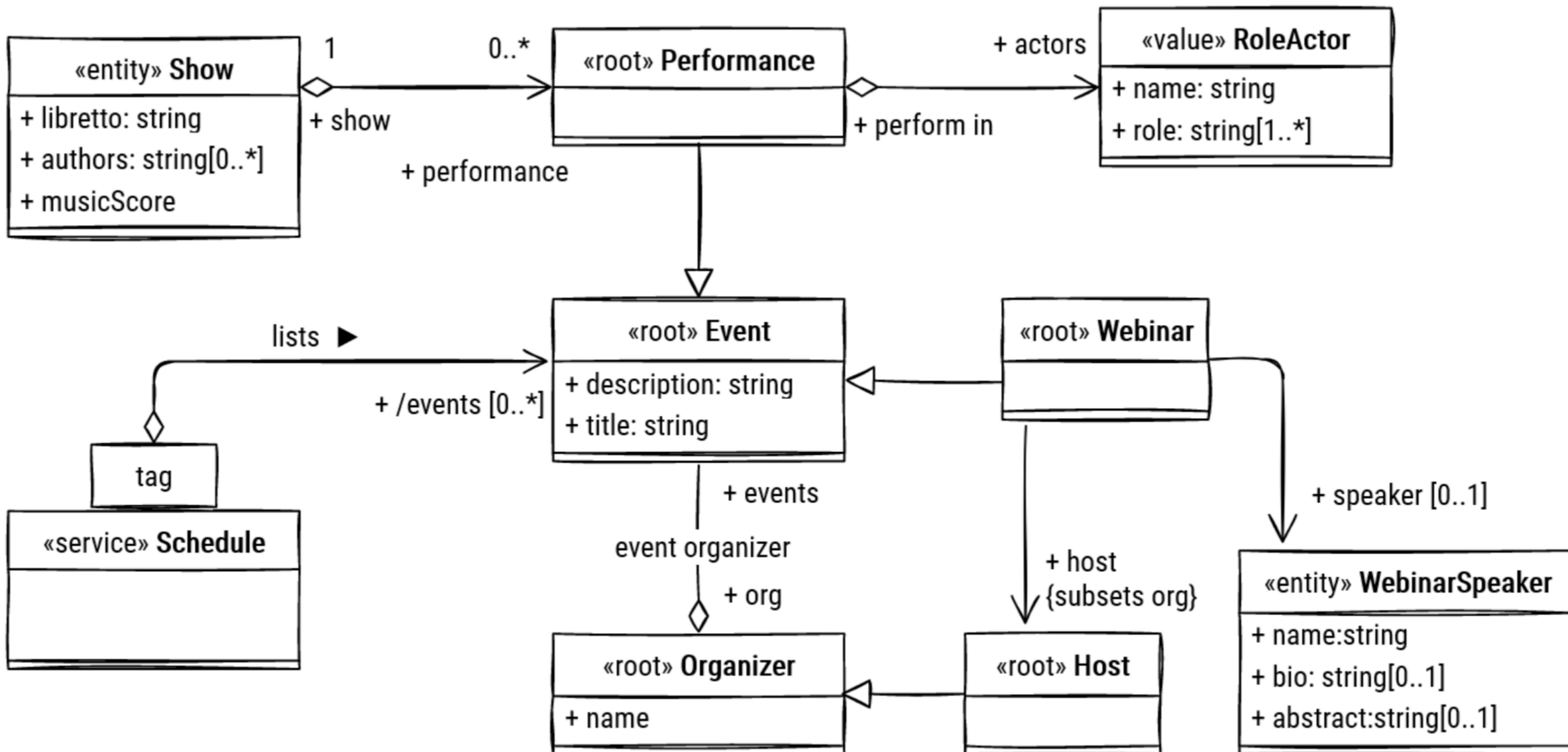
Сервисы предметной области (Domain Services) – это операции, которые не принадлежат ни к одной сущности или объекту-значению, но являются частью модели предметной области.



Строительные блоки DDD. Фабрики

Фабрики (Factories) – это объекты, отвечающие за создание сложных объектов или агрегатов.

Пример для билетной кассы с DDD стереотипами



Анализ текущего проекта

Где мы соответствуем принципам DDD?

1. Использование единого языка:

- Наши классы и методы используют термины из предметной области: `Car`, `Customer`, `CarService`, `HseCarService`.
- Комментарии к коду также используют термины из предметной области.

2. Четкое разделение на модули:

- Проект разделен на модули по функциональности: `Cars`, `Customers`, `Sales`, `Reports`, `Accounting`.
- Каждый модуль отвечает за свою часть функциональности.

3. Использование фабрик:

- `PedalCarFactory` и `HandCarFactory` отвечают за создание автомобилей разных типов.

4. Использование сервисов предметной области:

- `HseCarService` отвечает за бизнес-логику продажи автомобилей.

5. Инкапсуляция:

- Сущности `Car` и `Customer` инкапсулируют свое состояние и поведение.
- Некоторые свойства доступны только для чтения, что защищает инварианты.

Где мы противоречим принципам DDD?

1. Отсутствие явных агрегатов:

- В проекте нет явного определения агрегатов и их корней.
- Отношения между сущностями не всегда четко определены.

2. Смешение ответственостей:

- **CarService** выполняет функции как репозитория, так и сервиса предметной области.
- **CustomersStorage** также смешивает ответственности.

3. Отсутствие объектов-значений:

- В проекте не используются объекты-значения, хотя они могли бы упростить модель.

4. Недостаточное использование инвариантов:

- Бизнес-правила не всегда явно выражены в коде.
- Некоторые инварианты могут быть нарушены из-за отсутствия проверок.

5. Отсутствие событий предметной области:

- В проекте не используются события предметной области, что затрудняет отслеживание изменений.

6. Фасад как антипаттерн:

- **CarShop** является фасадом, который скрывает сложность системы, но также скрывает и модель предметной области.
- Фасад может противоречить принципам DDD, если он скрывает важные бизнес-концепции.

Превращаем проект в DDD-compliant

1. Ввести объекты-значения

- **EngineSpecification**: Объект-значение для хранения характеристик двигателя.
- **CustomerCapabilities**: Объект-значение для хранения возможностей покупателя (сила ног, сила рук).

Создание объекта-значения CustomerCapabilities

```
// Создаем новый файл CustomerCapabilities.cs в папке Customers
namespace UniversalCarShop.Customers;

public sealed record CustomerCapabilities
{
    public int LegPower { get; }
    public int HandPower { get; }

    public CustomerCapabilities(int legPower, int handPower)
    {
        if (legPower < 0)
            throw new ArgumentException("Сила ног не может быть отрицательной", nameof(legPower));

        if (handPower < 0)
            throw new ArgumentException("Сила рук не может быть отрицательной", nameof(handPower));

        LegPower = legPower;
        HandPower = handPower;
    }
}
```

Создание объекта-значения для двигателя

```
// Создаем новый файл EngineSpecification.cs в папке Engines
namespace UniversalCarShop.Energies;

public sealed record EngineSpecification
{
    public EngineSpecification(int requiredLegPower, int requiredHandPower)
    {
        if (requiredLegPower < 0)
            throw new ArgumentException("Требуемая сила ног не может быть отрицательной", nameof(requiredLegPower));

        if (requiredHandPower < 0)
            throw new ArgumentException("Требуемая сила рук не может быть отрицательной", nameof(requiredHandPower));

        RequiredLegPower = requiredLegPower;
        RequiredHandPower = requiredHandPower;
    }

    public int RequiredLegPower { get; }
    public int RequiredHandPower { get; }

    public bool IsCompatibleWith(CustomerCapabilities capabilities)
    {
        if (capabilities == null)
            throw new ArgumentNullException(nameof(capabilities));

        return capabilities.LegPower >= RequiredLegPower &&
               capabilities.HandPower >= RequiredHandPower;
    }
}
```

Рефакторинг классов для использования объектов-значений

```
// Модифицируем класс Customer
public sealed class Customer
{
    public Customer(string name, CustomerCapabilities capabilities)
    {
        if (string.IsNullOrWhiteSpace(name))
            throw new ArgumentException("Имя не может быть пустым", nameof(name));

        Name = name;
        Capabilities = capabilities ?? throw new ArgumentNullException(nameof(capabilities));
    }

    public string Name { get; }
    public CustomerCapabilities Capabilities { get; }
    public Car? Car { get; private set; }

    // Метод для назначения автомобиля покупателю
    public void AssignCar(Car car)
    {
        Car = car;
    }

    // Переопределяем ToString
    public override string ToString()
    {
        var builder = new StringBuilder();
        builder.Append($"Имя: {Name}. Сила ног: {Capabilities.LegPower}. Сила рук: {Capabilities.HandPower}. ");
        if (Car is null)
        {
            builder.Append("Автомобиль: { Нет }");
        }
        else
        {
            builder.Append($"Автомобиль: {{ {Car} }}");
        }
        return builder.ToString();
    }
}
```

Выделение агрегатов

```
// Модифицируем класс Car
public sealed class Car
{
    private readonly IEngine _engine;

    public Car(IEngine engine, int number)
    {
        Number = number;
        _engine = engine;
    }

    public int Number { get; }
    public bool IsSold { get; private set; }

    public bool IsCompatible(Customer customer) => _engine.IsCompatible(customer);

    public void MarkAsSold() => IsSold = true;

    public override string ToString() => $"Номер: {Number}. Двигатель: {_engine}";
}
```

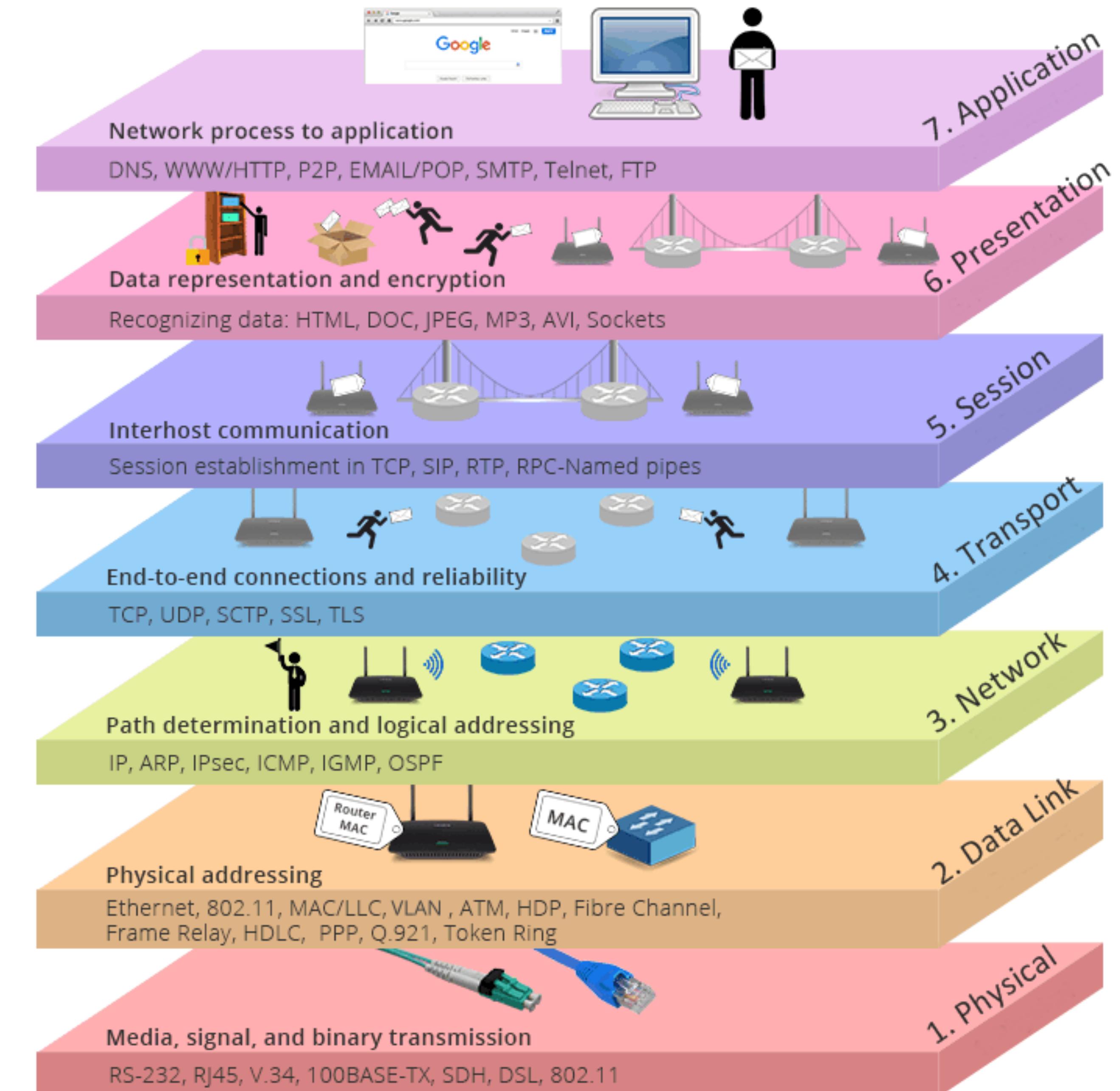
Networking

OSI, TCP, HTTP



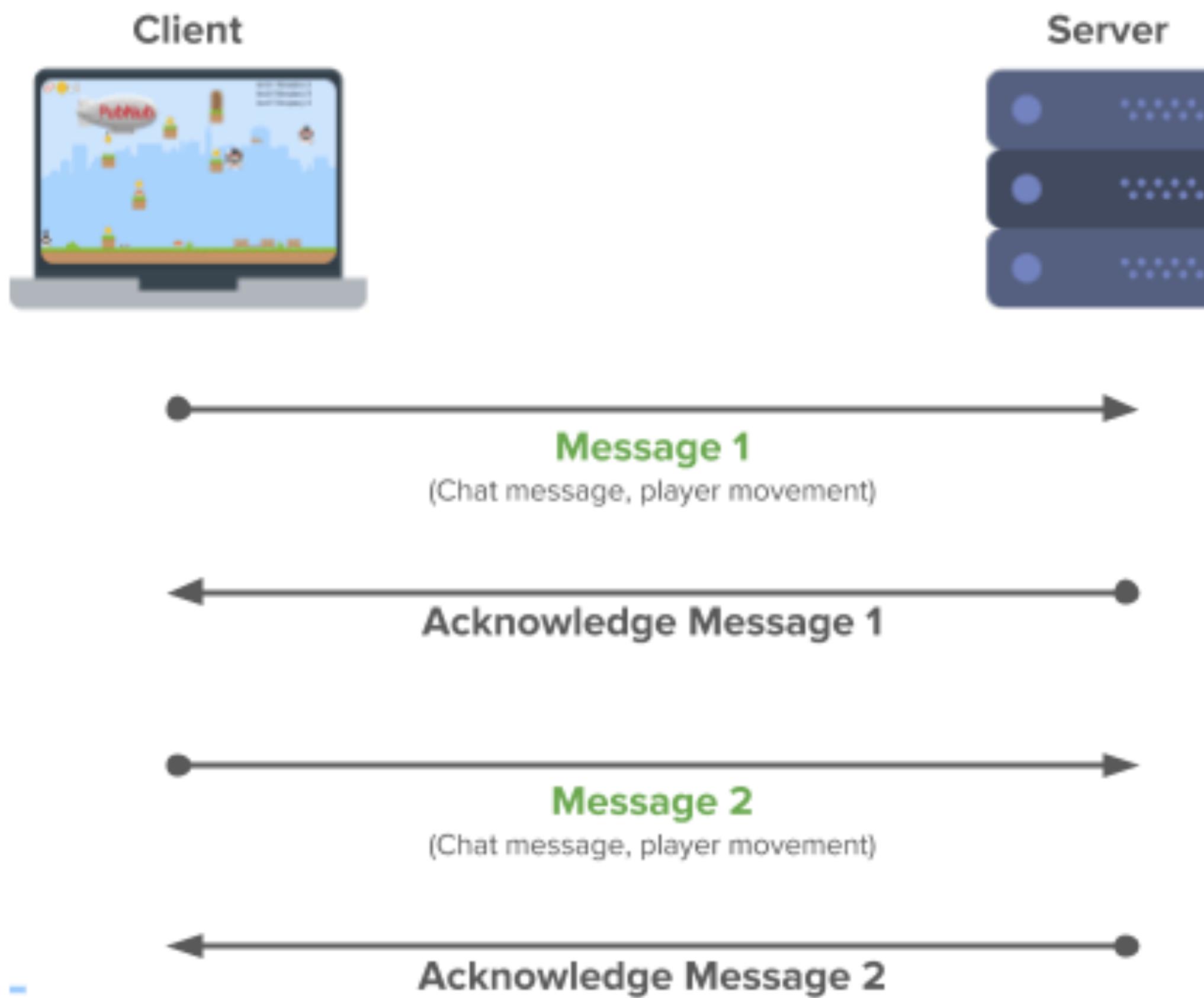
Модель OSI

1. **Физический уровень** – передача битов по каналу связи.
2. **Канальный уровень** – обеспечение надежной передачи данных.
3. **Сетевой уровень** – маршрутизация пакетов (IP, ICMP).
4. **Транспортный уровень** – управление передачей данных (TCP, UDP).
5. **Сеансовый уровень** – управление сессиями связи.
6. **Представительский уровень** – преобразование форматов данных.
7. **Прикладной уровень** – взаимодействие с приложениями (HTTP, FTP).

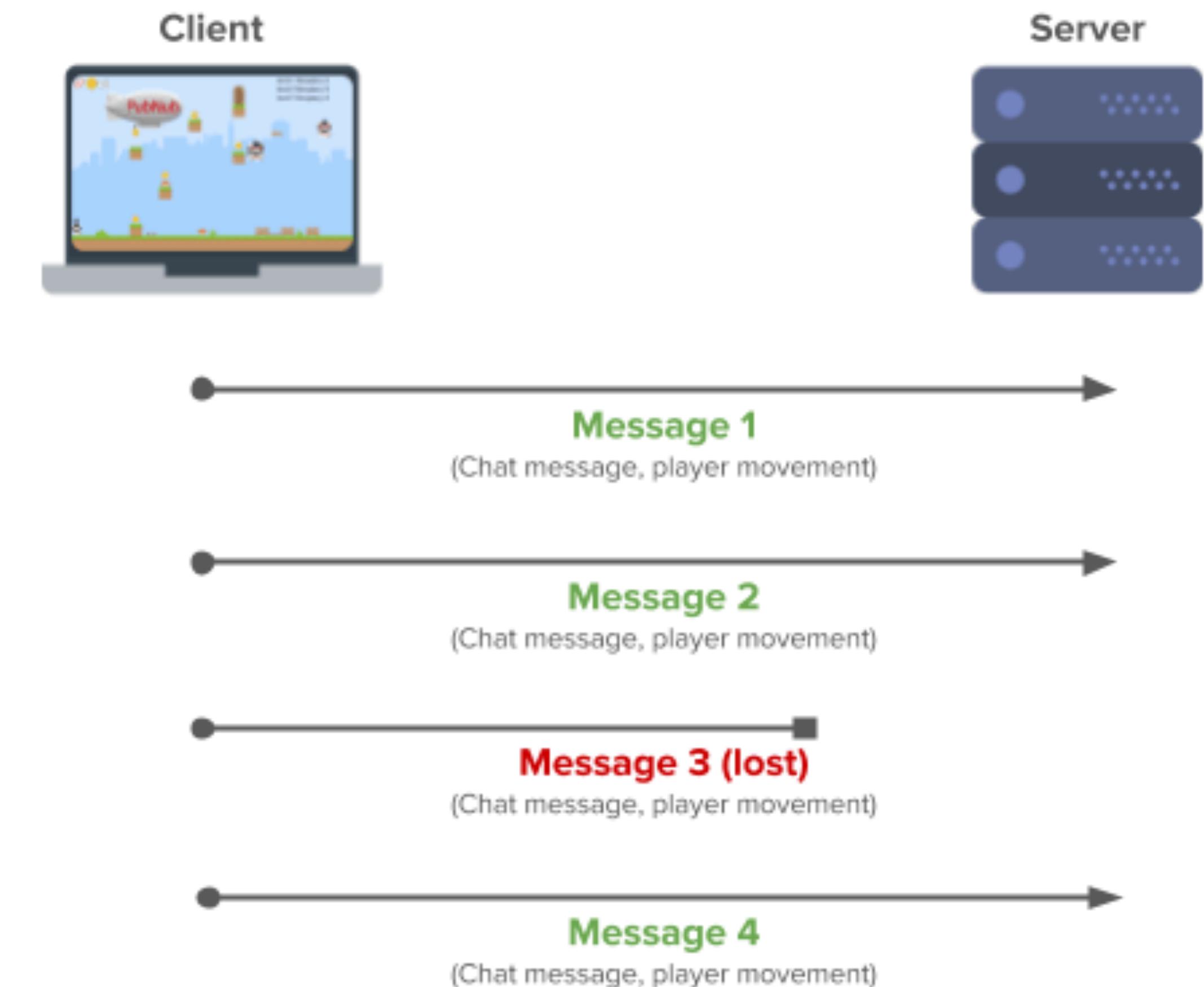


TCP vs UDP

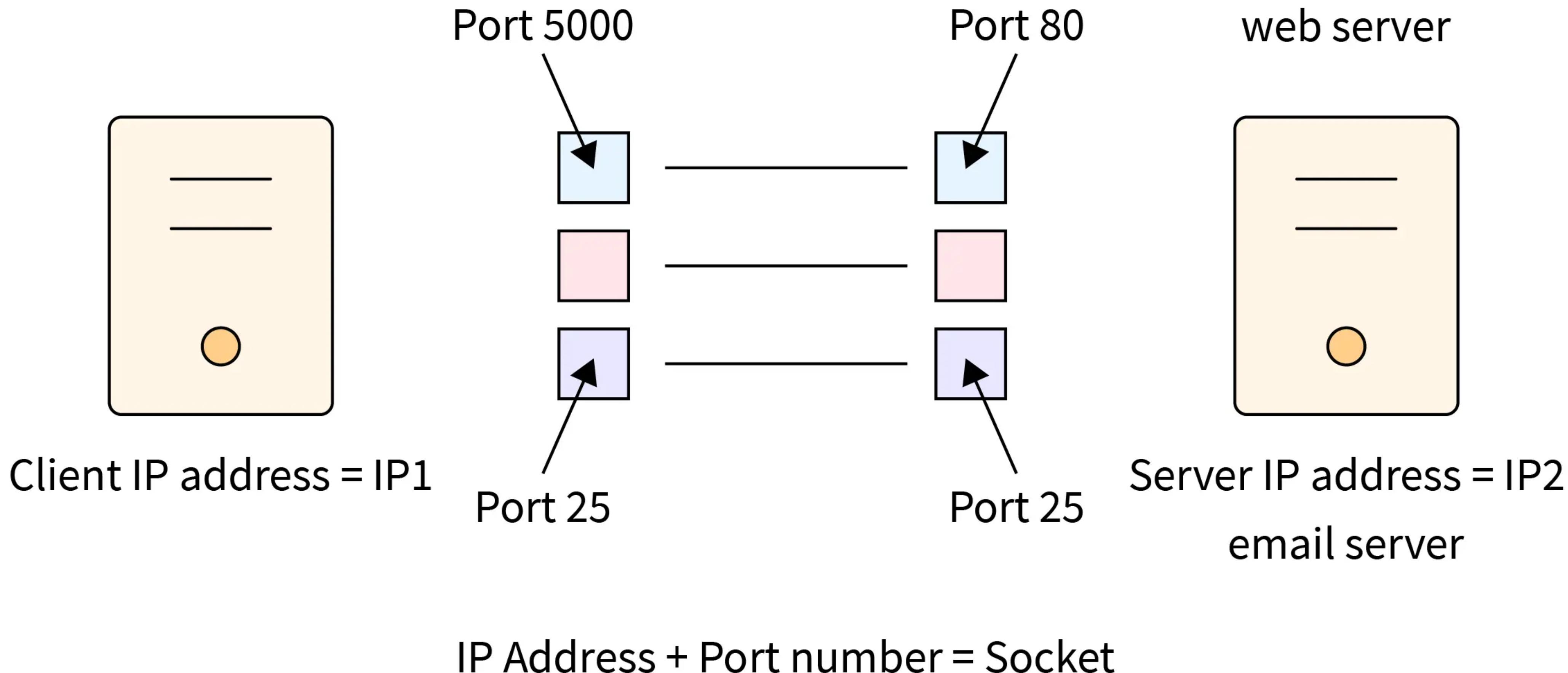
TCP



UDP

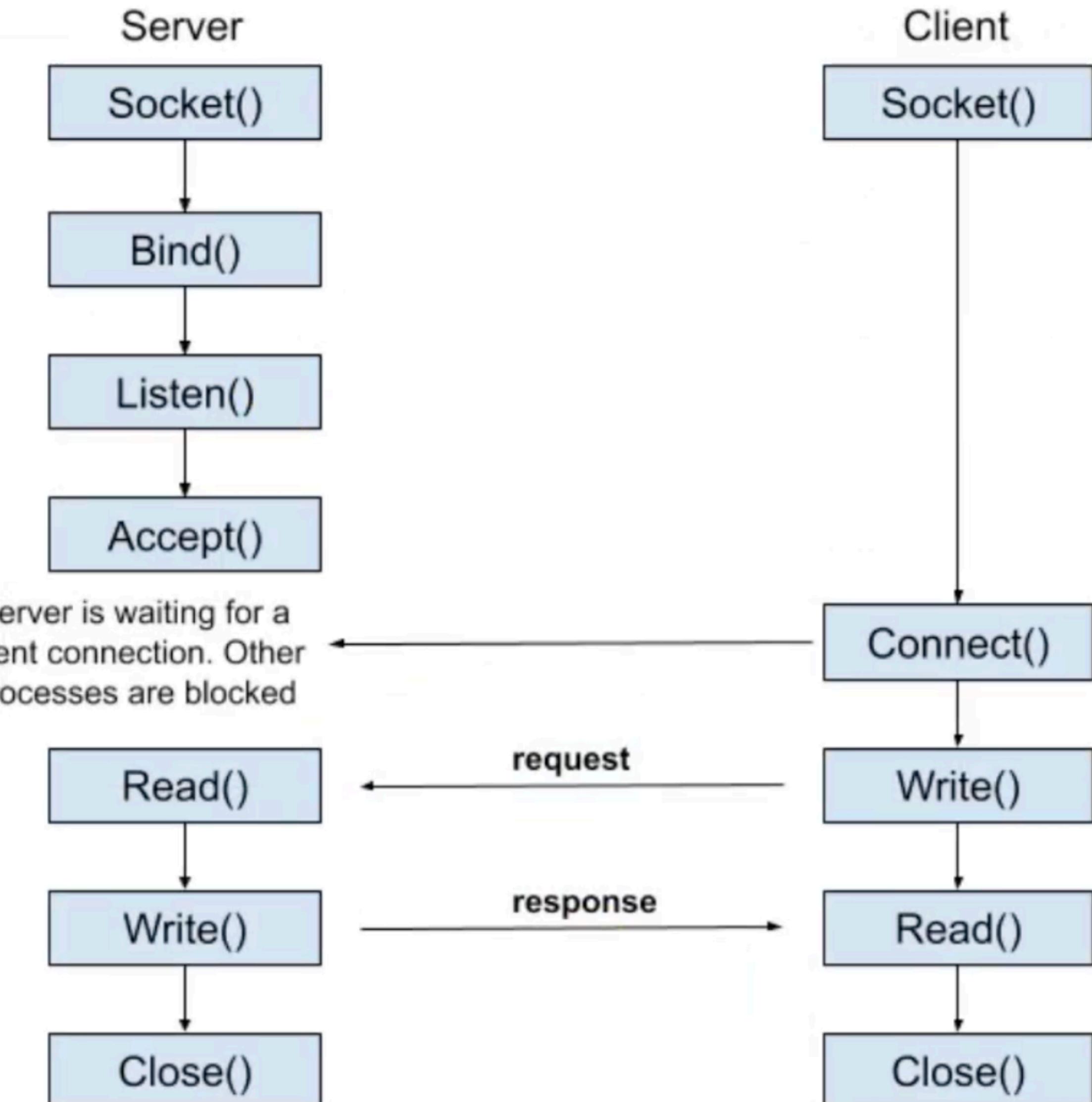


TCP/IP Port And Sockets



System.NET.Sockets

Позволит реализовать чтение/
запись в сокет



Пример использования Socket на клиенте

```
using System.Net.Sockets;
using System.Text;

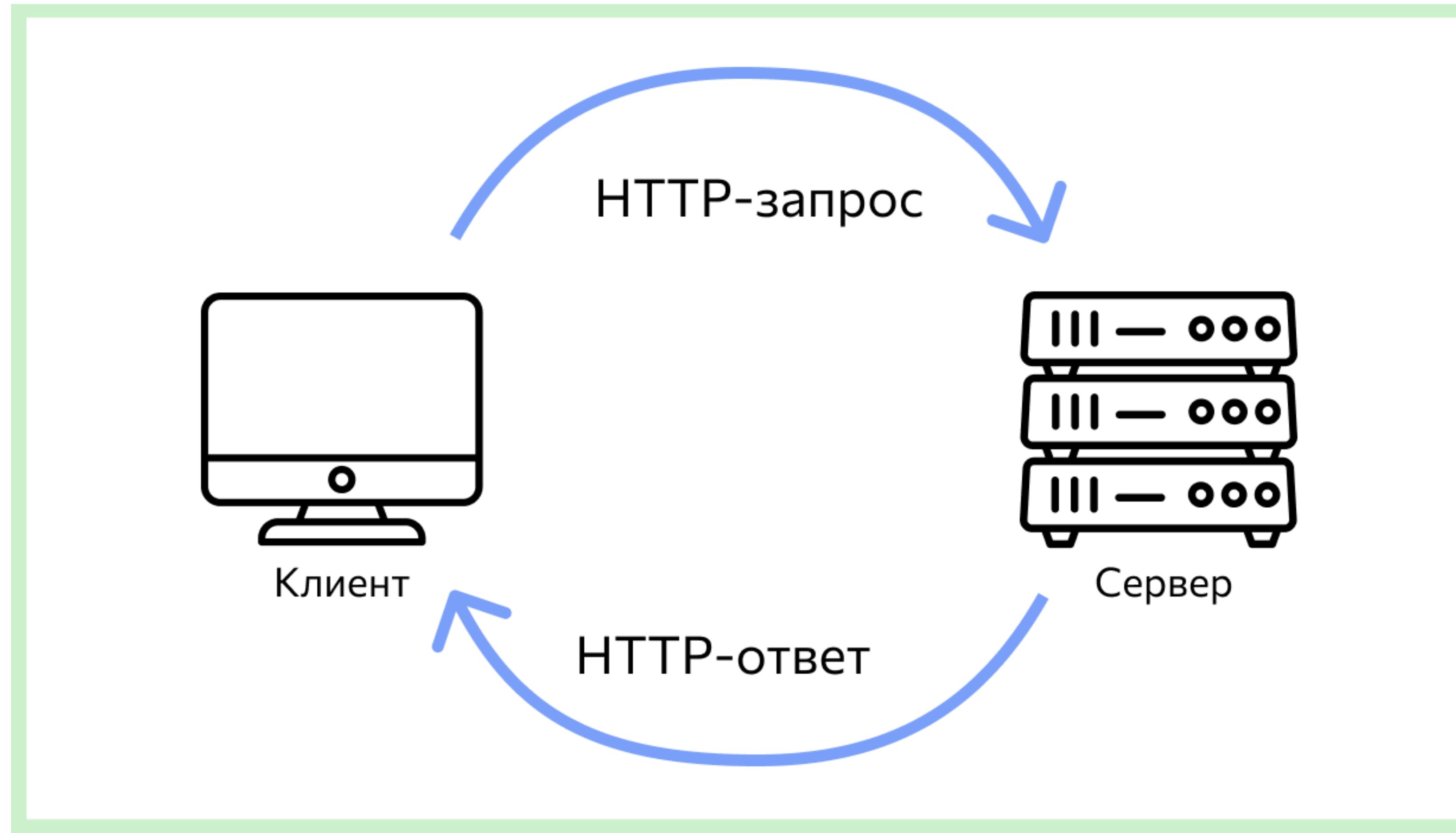
var socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream, ProtocolType.Tcp);
socket.Connect("hse.ru", 80);

string message = "Hello, HSE!";
byte[] dataToSend = Encoding.UTF8.GetBytes(message);
socket.Send(dataToSend);

byte[] buffer = new byte[1024];
int bytesRead = socket.Receive(buffer);
string response = Encoding.UTF8.GetString(buffer, 0, bytesRead);
Console.WriteLine(response);
```

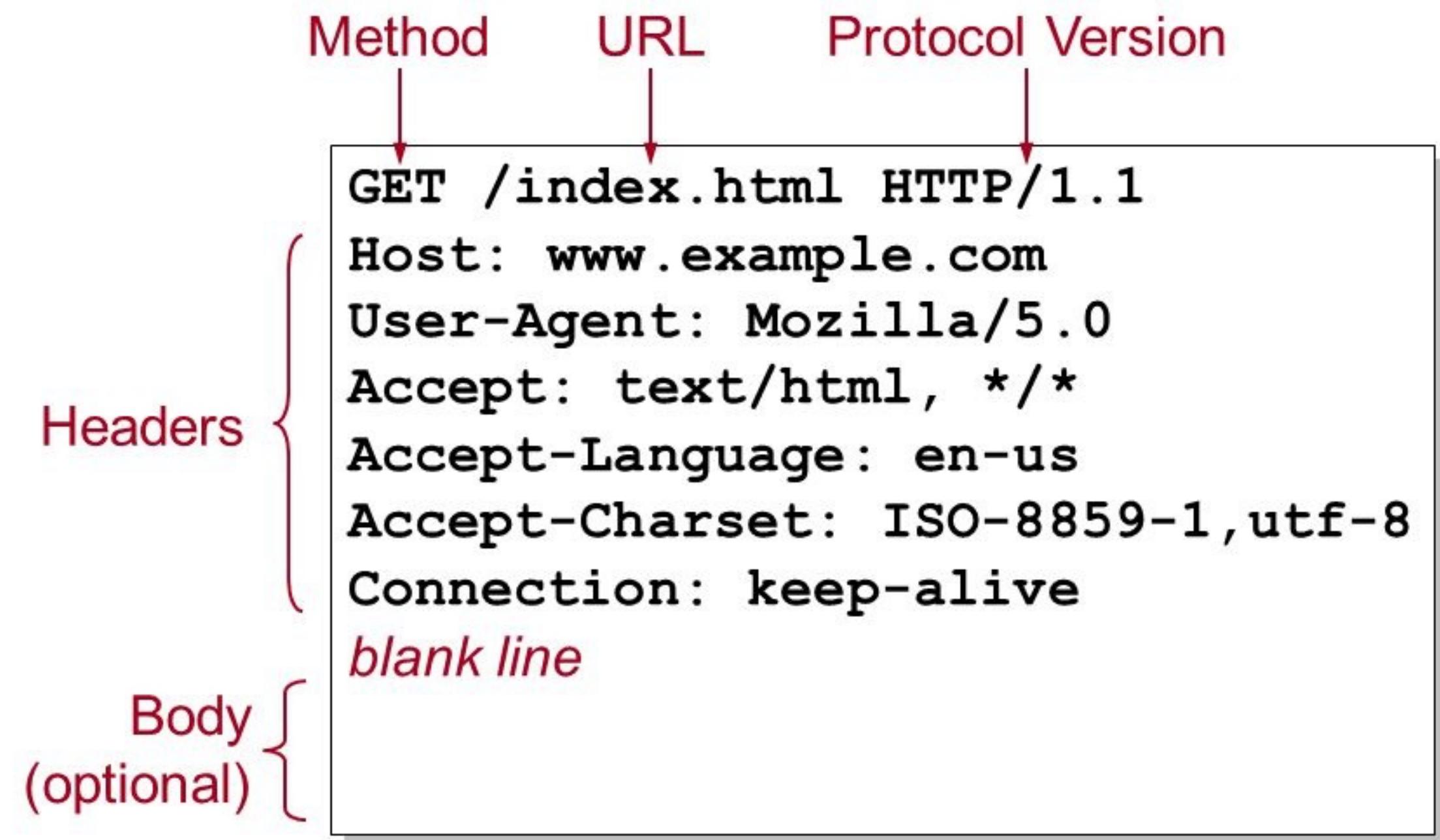
Здесь происходит создание сокета для установления TCP подключения, осуществляется передача данных, а также чтение ответа сервера

HTTP



HTTP Запросы

GET/POST/PATCH/PUT/DELETE



```
POST / HTTP/1.1
Host: localhost:8000
User-Agent: Mozilla/5.0 (Macintosh; ... )... Firefox/51.0
Accept: text/html,application/xhtml+xml,...,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Upgrade-Insecure-Requests: 1
Content-Type: multipart/form-data; boundary=-12656974
Content-Length: 345
```

-12656974
(more data)

Request headers
General headers
Representation headers

Response code

100 – 199 Информационные ответы

200 – 299 Успешное выполнение запроса

300 – 399 Редиректы

400 – 499 Клиентские ошибки

500 – 599 Ошибки на странице сервера

Пример с HttpClient

```
using HttpClient httpClient = new HttpClient();

try
{
    string uri = "https://cs.hse.ru";

    HttpResponseMessage response = await httpClient.GetAsync(uri);

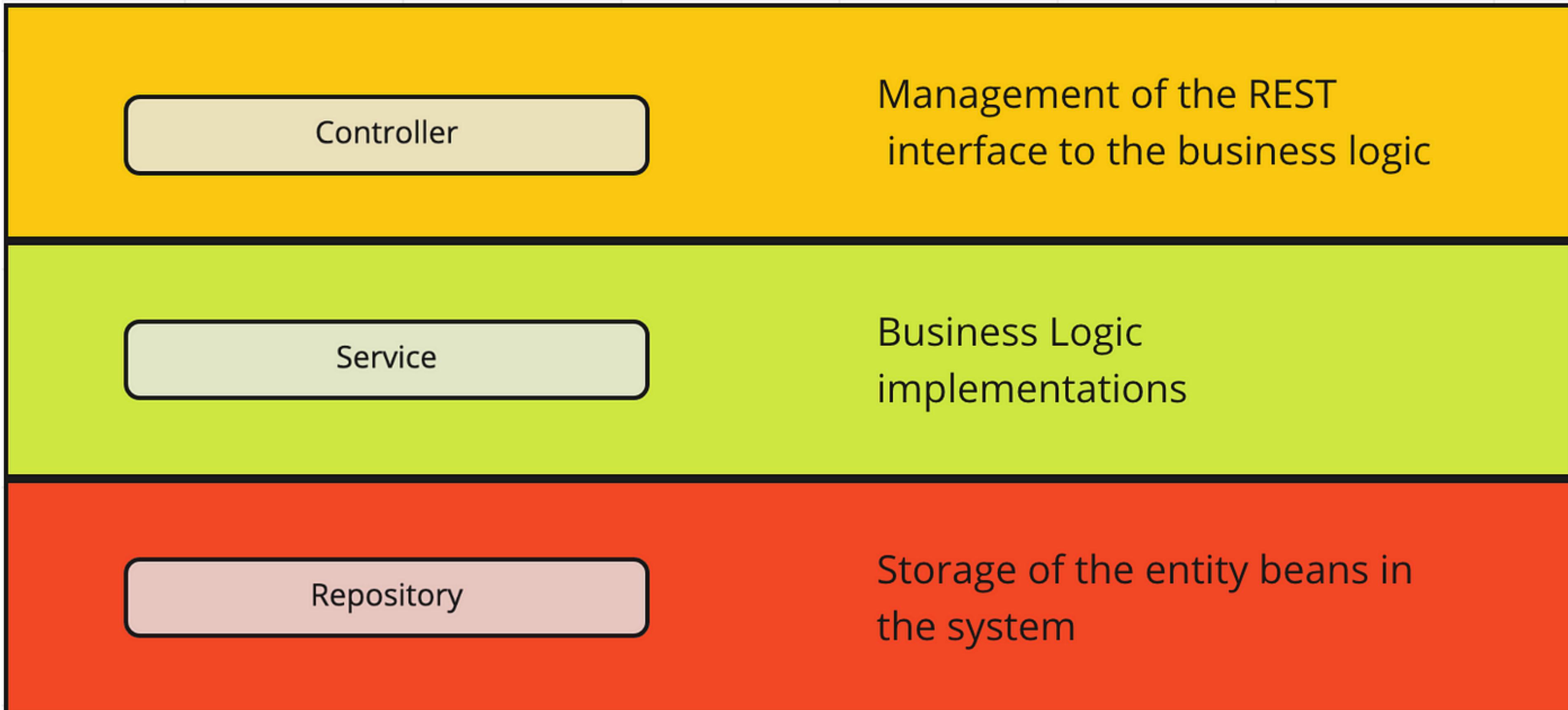
    response.EnsureSuccessStatusCode();

    string responseBody = await response.Content.ReadAsStringAsync();

    Console.WriteLine(responseBody);

    ...
}
```

Controller-Service-Repository



Веб-API

ASP.NET и Clean Architecture

ASP.NET Core

ASP.NET Core представляет собой современный, высокопроизводительный фреймворк для создания веб-приложений, который работает на всех основных платформах (Windows, Linux, macOS).

В отличие от своего предшественника, ASP.NET Framework, новая версия полностью переработана для обеспечения модульности и производительности.

Преимущества ASP.NET Core:

- **Кросс-платформенность:** Приложения могут разрабатываться и запускаться на Windows, Linux и macOS
- **Модульная архитектура:** Приложения включают только необходимые зависимости
- **Встроенный контейнер DI (Dependency Injection):** Упрощает тестирование и поддерживает слабосвязанную архитектуру
- **Высокая производительность:** Один из самых быстрых веб-фреймворков согласно TechEmpower benchmarks
- **Единый стек для веб-UI и веб-API:** Разработка различных типов веб-приложений с использованием общих концепций

Контроллеры vs Minimal API

Минимальный API (Minimal API):

- Введен в ASP.NET Core 6.0 как новый, упрощенный способ создания API
- Сокращает шаблонный код (boilerplate) и уменьшает количество файлов
- Все взаимодействие настраивается непосредственно в Program.cs
- Идеально подходит для микросервисов и небольших проектов
- Оптимизирован для производительности с меньшим использованием рефлексии

```
app.MapGet("/weatherforecast", () =>
{
    var forecast = Enumerable.Range(1, 5).Select(index =>
        new WeatherForecast
        (
            DateOnly.FromDateTime(DateTime.Now.AddDays(index)),
            Random.Shared.Next(-20, 55),
            summaries[Random.Shared.Next(summaries.Length)])
    )
    .ToArray();
    return forecast;
})
.WithName("GetWeatherForecast")
.WithOpenApi();
```

Контроллеры vs Minimal API

Controller-based API:

- Хорошо структурированная модель с разделением ответственности
- Основан на контроллерах, которые группируют связанные конечные точки (endpoints)
- Поддерживает полный набор функций MVC (фильтры, связывание модели, валидация)
- Следует устоявшимся практикам и шаблонам REST
- Существует с начала ASP.NET Core 1.0

```
[ApiController]
[Route("[controller]")]
public class WeatherForecastController : ControllerBase
{
    private readonly ILogger<WeatherForecastController> _logger;

    public WeatherForecastController(ILogger<WeatherForecastController> logger)
    {
        _logger = logger;
    }

    [HttpGet(Name = "GetWeatherForecast")]
    public IEnumerable<WeatherForecast> Get()
    {
        // метод контроллера
    }
}
```

Когда использовать Minimal API

Minimal API лучше подходит для определенных сценариев:

Микросервисы и небольшие проекты

- Упрощенная структура проекта с меньшим количеством файлов
- Более быстрая разработка благодаря меньшему количеству шаблонного кода
- Легче понять весь сервис, так как весь код находится в одном месте

Производительно-критичные приложения

- Меньше накладных расходов на обработку запросов
- Меньше рефлексии во время выполнения

Пример сценария: Микросервис для обработки платежей с несколькими четко определенными эндпоинтами будет более эффективен и прост в реализации с помощью Minimal API.

Когда использовать традиционный API

Традиционный подход с контроллерами является более зрелым и предоставляет ряд преимуществ в определенных сценариях:

Сложные бизнес-логики и большие проекты

- Лучшая организация кода в больших проектах с множеством эндпоинтов
- Естественное группирование связанных эндпоинтов в контроллеры
- Более чёткое разделение ответственности

Расширенные возможности MVC

- Полная поддержка фильтров (авторизация, кэширование, валидация)
- Более мощные механизмы привязки моделей
- Встроенная поддержка валидации с помощью атрибутов

Пример сценария: Корпоративное приложение с множеством взаимосвязанных бизнес-сущностей и сложной логикой авторизации будет лучше структурировано с использованием контроллеров.

REST

REST – это архитектурный стиль для проектирования распределенных систем, особенно веб-сервисов. REST не связан с конкретной технологией и может быть реализован на любой платформе.

REST

Основные принципы REST:

- **Ресурсы:** Данные и функциональность представлены как ресурсы, которые идентифицируются через URI (Uniform Resource Identifier).
- **HTTP методы:** Для операций с ресурсами используются стандартные HTTP методы (GET, POST, PUT, DELETE).
- **Представления:** Ресурсы могут иметь различные представления (JSON, XML, HTML).
- **Отсутствие состояния:** Каждый запрос содержит всю необходимую информацию для его выполнения.
- **HATEOAS** (Hypermedia as the Engine of Application State): API предоставляет гиперссылки, которые клиент может использовать для перехода между ресурсами.

Примеры RESTful API:

- GET /tasks — получить список задач
- GET /tasks/123 — получить информацию о конкретной задаче
- POST /tasks — создать новую задачу
- PUT /tasks/123 — обновить существующую задачу
- DELETE /tasks/123 — удалить задачу

MVC

MVC — это шаблон проектирования, который разделяет приложение на три основных компонента, обеспечивая разделение ответственности.

Компоненты MVC:

- **Модель (Model):** Представляет данные и бизнес-логику приложения. Модель уведомляет представления об изменениях данных.
- **Представление (View):** Отвечает за визуализацию данных для пользователя. Получает данные от модели и отображает их.
- **Контроллер (Controller):** Обрабатывает входящие запросы, взаимодействует с моделью и выбирает представление для ответа.

В контексте ASP.NET Core API, мы часто используем только компоненты Model и Controller (без View), поскольку API обычно возвращает данные в форматах JSON или XML, а не HTML-страницы.

Clean Architecture

