??????????? ???????? ?????? ?? ???-?????? ? ?????? ????????? ???????

???? ???????????

Архитектура доставки сообщений должна:

- поддерживать несколько соединений на одного пользователя (несколько вкладок)
- обеспечивать разные модели доставки:
  - unicast (одному пользователю)
  - multicast (группе пользователей)
  - broadcast (всем)
- эффективно работать при большом количестве сообщений
- изолировать медленные клиенты (backpressure)
- быть расширяемой без переписывания существующего кода

????? ????? ???????????

```
MessageSource
  ↓
TransportSerializer (сериализация в JSON)
  ↓
MessageEnvelop
  ↓
MessageRouter
  ↓
DeliveryStrategy
  ↓
ConnectionRegistry
  ↓
ConnectionContext (BoundedChannel)
```

```
  ↓
SendLoop
  ↓
WebSocket
```

MessageEnvelop

Все сообщения передаются в едином формате-обёртке:

```
public sealed class MessageEnvelope
{
    public DeliveryType DeliveryType { get; init; }

    public Guid? UserId { get; init; }                    // для Unicast
    public IEnumerable<Guid>? UserIds { get; init; }      // для Multicast

    public ReadOnlyMemory<byte> Payload { get; init; }    // готовые байты (JSON)
}

public enum DeliveryType
{
    Unicast,
    Multicast,
    Broadcast
}
```

> На уровне WebSocket-доставки используются только готовые байтовые представления сообщений, что полностью изолирует транспортный слой от формата данных.

TransportSerializer

Доменные моедли сериализуются через `TransportSerializer`:

```
public static class TransportSerializer
{
```

```
    public static ReadOnlyMemory<byte> Serialize(OutgoingMessage message)
    {
        var typeName = message.GetType().Name;

        var transportObject = new
        {
            type = typeName,
            data = (object)message
        };

        return JsonSerializer.SerializeToUtf8Bytes(transportObject);
    }
}

// Example
var message = new UnitStatusPack
{
    UnitId = id,
    Status = "Working",
    Progress = 42
};

var payload = TransportSerializer.Serialize(message);

var envelope = new MessageEnvelope
{
    DeliveryType = DeliveryType.Broadcast,
    Payload = payload
};
```

> Сериализация происходит до создания MessageEnvelope. Т.к. транспортный слой работает только с байтами.

MessageRouter

Ответственность заключается в выборе стратегии доставки по типу сообщения.

> Не знает ничего о реализации отправки, что способствует для последующией миграции на другие протоколы.

```
public sealed class MessageRouter : IMessageRouter
{
    private readonly IReadOnlyDictionary<DeliveryType, IMessageDeliveryStrategy> _strategies;

    public Task RouteAsync(MessageEnvelope message)
    {
        if (!_strategies.TryGetValue(message.DeliveryType, out var strategy))
            throw new InvalidOperationException($"No strategy for {message.DeliveryType}");

        return strategy.SendAsync(message);
    }
}
```

Delivery Strategy

Каждая модель доставки реализуется отдельной стратегией.

```
public interface IMessageDeliveryStrategy
{
    Task SendAsync(MessageEnvelope message);
}
```

Стратегия	Назначение	Параметры MessageEnvelope
UnicastStrategy	Сообщение одному пользователю	UserId
MulticastStrategy	Группе пользователей	UserIds
BroadcastStrategy	Всем подключениям	-

Реализация:

- Стратегии получают список соединений из ConnectionRegistry
- Проверяют состояние WebSocket перед отправкой
- Записывают сообщение в Channel каждого соединения через TryWrite
- Логируют результаты доставки (успешные/неуспешные)

>

> Если `TryWrite` возвращает false (канал полон или закрыт), сообщение логируется как неуспешное, но не блокирует доставку другим соединениям.

ConnectionRegistry

Единый источник информации о подключениях.

```
public interface IConnectionRegistry
{
    void Add(ConnectionContext ctx);
    void Remove(ConnectionContext ctx);

    IEnumerable<ConnectionContext> GetUserConnections(Guid userId);
    IEnumerable<ConnectionContext> GetUsersConnections(IEnumerable<Guid> userIds);
    IEnumerable<ConnectionContext> GetAllConnections();
}
```

> Один пользователь может иметь несколько активных соединений.

Жизненный цикл:

- Подключение:
  - Создаётся `ConnectionContext` для WebSocket
  - Вызывается `ConnectionRegistry.AddConnection(ctx)`
  - Обновляются структуры _userConnections, _allConnections
- Отключение:
  - WebSocket закрыт или исключение в SendLoop/RecieveLoop
  - Вызывается `ConnectionRegistry.RemoveConnection(ctx)`
  - Удаляется из всех словарей

ConnectionContext

Каждое WebSocket-соединение оборачивается в `ConnectionContext`:

```
public sealed class ConnectionContext
{
```

```
    public Guid ConnectionId { get; } = Guid.NewGuid();
    public Guid UserId { get; }
    public WebSocket Socket { get; }

    public Channel<ReadOnlyMemory<byte>> Outgoing { get; }
}
```

Назначение:

- изолировать медленные соединения
- обеспечить backpressure
- избежать блокировки при массовой рассылке

??????? ?????????? (backpressure)

Используется bounded channel с настраиваемым размером:

```
Channel.CreateBounded<ReadOnlyMemory<byte>>(
    new BoundedChannelOptions(channelBufferSize)  // из конфигурации, по умолчанию 100
    {
        FullMode = BoundedChannelFullMode.DropOldest
    }
);
```

Поведение:

- если клиент не успевает читать → старые сообщения отбрасываются
- сервер не растёт по памяти

> **Важно:** С `BoundedChannelFullMode.DropOldest`:
>
> - Если канал полон → `TryWrite` возвращает `true`, новое сообщение записывается, **старое сообщение вытесняется и теряется**
> - Если канал закрыт → `TryWrite` возвращает `false`, сообщение не записывается и логируется как неуспешное
>
> Таким образом, стратегии **не блокируются** при полном канале, но **старые сообщения теряются** для медленных клиентов. Это обеспечивает изоляцию медленных соединений и предотвращает рост памяти сервера.

SendLoop

Для каждого WebSocket-соединения создаётся независимый асинхронный цикл отправки (SendLoop), который читает сообщения из ограниченной очереди. Это обеспечивает изоляцию клиентов, предотвращает блокировку при массовой рассылке и защищает сервер от неконтролируемого роста памяти.

```
public static async Task RunAsync(
    ConnectionContext ctx,
    CancellationToken cancellationToken)
{
    var socket = ctx.Socket;
    var reader = ctx.Outgoing.Reader;

    try
    {
        while (!cancellationToken.IsCancellationRequested &&
               socket.State == WebSocketState.Open)
        {
            var payload = await reader.ReadAsync(cancellationToken);

            await socket.SendAsync(
                payload,
                WebSocketMessageType.Text,
                endOfMessage: true,
                cancellationToken);
        }
    }
    catch (OperationCanceledException)
    {
        // нормальный shutdown
    }
    catch (ChannelClosedException)
    {
        // канал закрыт
    }
    catch (WebSocketException)
    {
        // клиент отвалился
    }
```

```
    finally
    {
        await SafeCloseAsync(socket);
    }
}
```

- Если сообщений нет → поток освобождён
- Loop “просыпается” только когда появляется сообщение
- Медленный клиент блокирует только свой SendLoop
- Массовая рассылка не ждёт `await SendAsync` других соединений

?????????? ???? ???????????

Подключение:

1. HTTP → WebSocket upgrade
2. JWT Аутентификация пользователя
3. Создание ConnectionContext
4. Регистрация в ConnectionRegistry
5. Запуск SendLoop

?????????? ??????????????

???????? ????? ??? ????????:

1. Создать новую стратегию, реализующую IMessageDeliveryStrategy
2. Добавить новый DeliveryType в enum
3. Зарегистрировать стратегию в DI
4. Обновить MessageRouter для маппинга типа на стратегию

???????? ????????? ???????? ?????????

- Расширить WaitForCloseAsync для обработки текстовых сообщений
- Добавить десериализацию и обработку команд от клиента

???????? ????????? UnitConnections

- Расширить ConnectionContext полем UnitId
- Добавить методы `GetUnitConnections` в `ConnectionRegistry`
- Обновить стратегии для фильтрации по UnitId

Revision #4
Created 2026-01-23 03:44:17 UTC by Fedmog1lnkv
Updated 2026-01-26 19:17:11 UTC by Fedmog1lnkv