

1. Ctesiphon
  1. Introduction
    1. Motivation
  2. Architecture
    1. System Overview
    2. Flow
      1. Inbound Task
      2. Outbound Task
        1. The Outbound Queue
  3. Implementation & Source Code
    1. Main
    2. Middleware
    3. Core
      1. State
      2. Chat Client
        1. Chat Client Handlers
  4. Prerequisites
    1. NET 5.0
    2. Redis
      1. Installing
      2. Redis-Cli
  5. Testing
    1. Subscribe
    2. Unsubscribe
    3. Message
      1. Note:
  6. Further developments

# Ctesiphon

---



Websockets



redis

# Introduction

---

This will be a multi-part series in which we are going to build from scratch a **Chat application** . In this article we are going to build the chat server which is the backbone of the chat application.

Supported Features:

- subscription to one or multiple channels
- unsubscription from target/all channels
- sending messages to target channel
- receiving messages from all subscribed channels

## Motivation

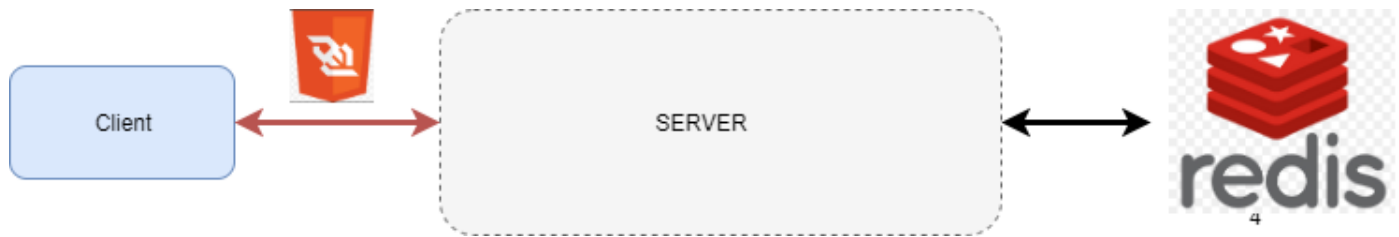
Ever since i started playing online games in middle-school back in 2003 (Warcraft 3) , i have been using messaging applications in order to communicate with my peers. The first such application which in time became ubiquitous was Skype.

Years after completely abandoning gaming and dabbling for some time in areas such as Industrial Automation , Embedded Devices i rediscovered my passion for messaging apps , but this time i was poised to create them.

# Architecture

---

## System Overview

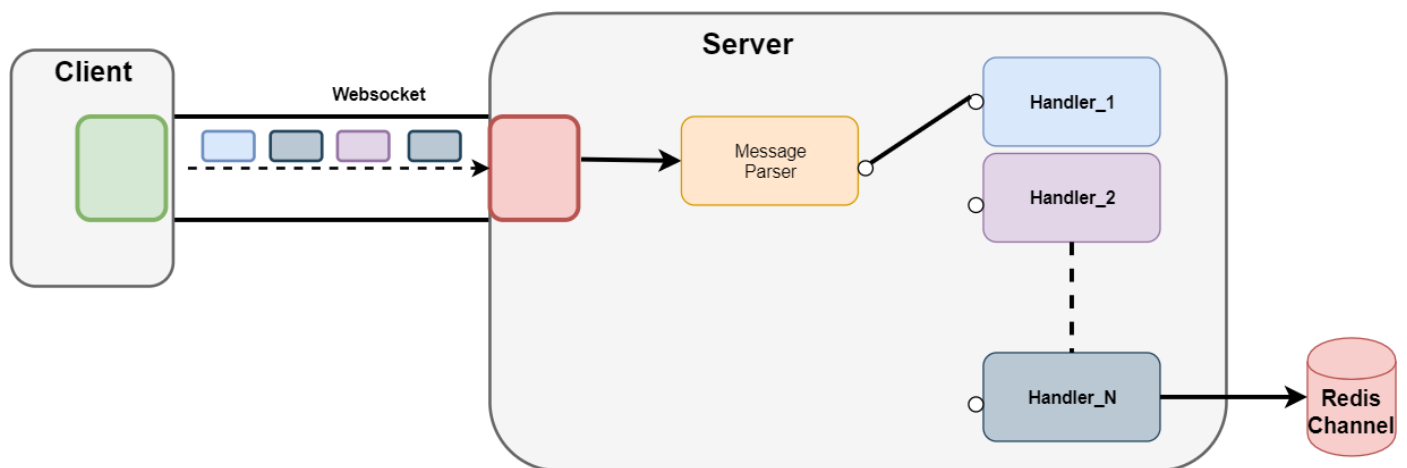


- **Server:** ASP NET Core Web application - the server where our logic will run handling client operations (subscribe/unsubscribe/publish message/get channels)
- **Database :** Redis as a message broker with its publish/subscribe functionality and also for storage (user subscribed channels)
- **Client Communication Protocol :** Since this is a chat application (bidirectional communication required) , the protocol we will be using is **Websockets**.

# Flow

By flow we will be referring to the way both inbound- messages arriving from the client and outbound messages sent to the client are handled and where and how does the Websocket object fit in as well as the Redis database.

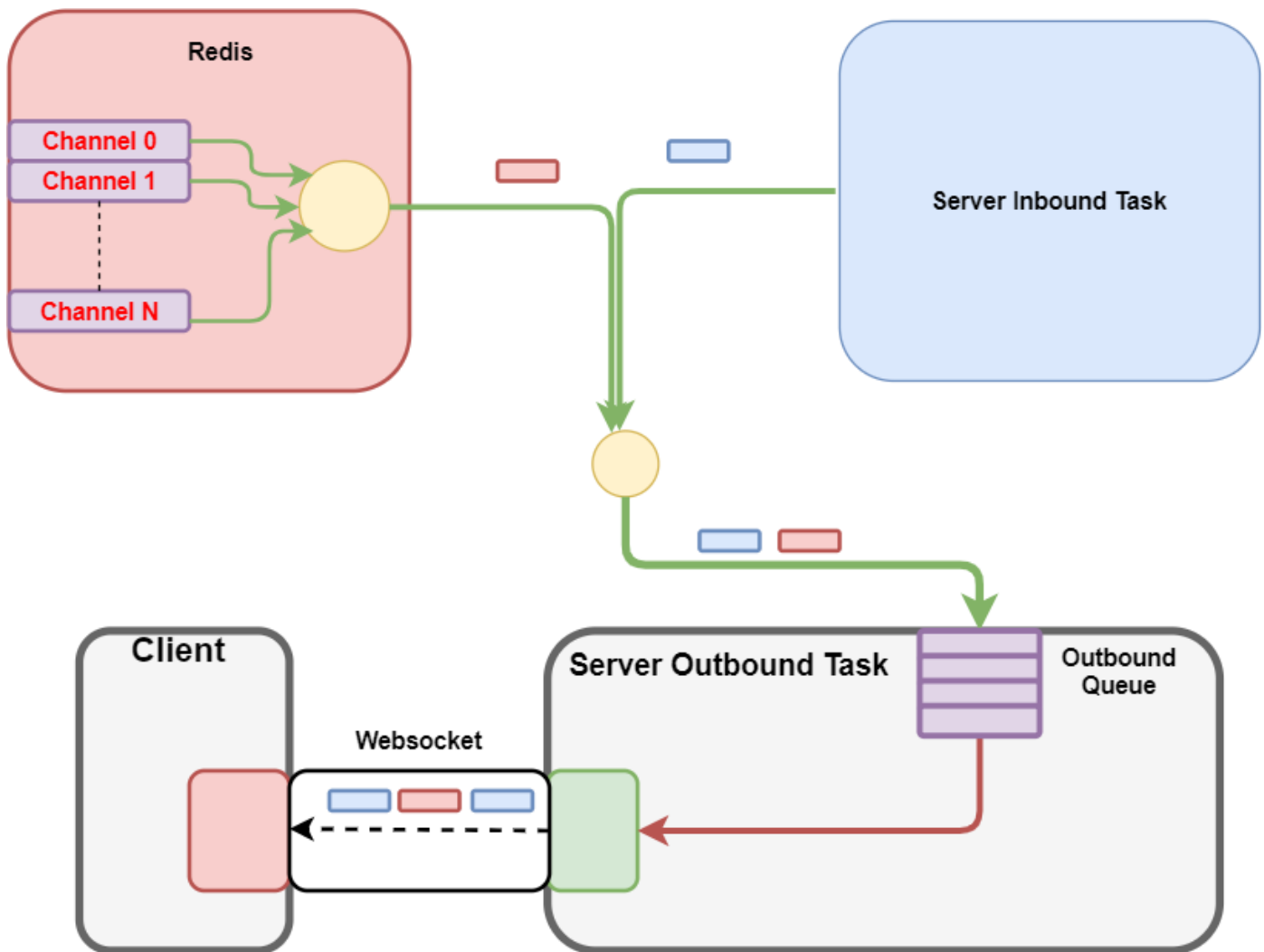
## Inbound Task



The inbound **Task** is basically a loop which receives messages from the client , parses them and dispatches them to an appropriate handler as can be seen from above.

**Note:** One such handler will write messages to a redis **channel**.

## Outbound Task



The outbound Task is a loop started asynchronously from the inbound task. Its purpose is to pop items off the queue and write them over the websocket to the connected client.

### The Outbound Queue

The outbound queue acts as a sink for all producers as can be seen from the picture. In our case the producers are:

- **Inbound Task:** The messages that the server sends back to the client (messages of type `SERVER_RESULT`)
- **Redis:** All messages that are published on channels on which our user is subscribed to.

As long as there are messages available in the queue we pop them and send them to the client over the websocket connection.

When there are no messages inside the queue, the task blocks, awaiting new ones.

# Implementation & Source Code

---

We will be starting our project from a template of type **ASP NET Core Web Application**.

## Main

```
/// The entrypoint in our application
public class Program {
    public static void Main(string[] args) {
        CreateWebHostBuilder(args).Build().Run();
    }

    public static IWebHostBuilder CreateWebHostBuilder(string[] args) {
        var webhostbuilder = WebHost.CreateDefaultBuilder(args)
            .UseUrls(Constants.SERVER_URL)
            .UseStartup<Startup>();
        return webhostbuilder;
    }
}
```

```
/// Called above by extension method `UseStartup`
/// Constants.REDIS_CONNECTION is a plain string , eg: localhost:6379 (6379 is
the default redis port)

public class Startup {

    public IConfiguration Configuration { get; }

    public Startup(IConfiguration configuration) {
        Configuration = configuration;
    }

    public void ConfigureServices(IServiceCollection services) {
        services.AddControllers();
        ConnectionMultiplexer mux =
ConnectionMultiplexer.Connect(Constants.REDIS_CONNECTION);
        services.AddSingleton(mux);
    }

    public void Configure(IApplicationBuilder app) {
        app.UseRouting();
        app.UseWebSockets();
        app.MapWhen(y => y.WebSockets.IsWebSocketRequest, a =>
a.UseMiddleware<SocketWare>());
    }
}
```

The above sections are mandatory in any **ASP NET Core** application and are automatically generated from the template .

The **Program.Main** starts the application and will use the **Startup** class to configure it.

In the **Startup.ConfigureServices** , the **ConnectionMultiplexer** is our connection to the redis database and will be injected as a singleton resource in our application. Connected clients will use the multiplexer as a factory for subscriptions to target channels.

## Middleware

In the **Startup** class **Configure** method above , notice the **MapWhen** extension . Based on a predicate it will route all requests to the specified **ASP NET Core Middleware**.

In our case:

- predicate = request should be of type websocket
- middleware is of type **SocketWare** (presented below)

```
public class SocketWare {
    private RequestDelegate next;
    private ConnectionMultiplexer mux;
    public SocketWare(RequestDelegate _next, ConnectionMultiplexer mux) {
        this.next = _next;
        this.mux = mux;
    }

    ///Called by the framework on each websocket request
    public async Task Invoke(HttpContext context) {
        using (var socket = await context.WebSockets.AcceptWebSocketAsync()) {
            ChatClient client = new ChatClient(this.mux);
            await client.RunAsync(socket);
        }
    }
}
```

The **ConnectionMultiplexer** is passed using dependency injection - remember it was injected in the **Startu.ConfigureServices** method !

The **Invoke** method is a minimal requirement for any **ASP NET Core Middleware** so that the framework knows to route the incoming request , and lets you handle it.



## Core

This is the core of the application and since it is the most complex part i will post the entire component , and will explain it afterwards.

### State

The core component uses a private field of type [State](#) for its operations.

```
internal class State {  
    public string ClientId { get; set; }  
    public Task outboundTask;  
    public ISubscriber subscriber;  
    public IDatabase redisDB;  
}
```

- [ISubscriber](#) is a component of [StackExchangeRedis](#) and is used to subscribe/unsubscribe on redis channels.
- [IDatabase](#) is also a component of [StackExchangeRedis](#) and is used for issuing redis queries.
- [outboundTask](#) - the task that runs the outbound flow ( taking messages from the queue and pushing them over the websocket)

### Chat Client

```
public sealed partial class ChatClient {  
  
    private const int BUFFER_SIZE = 1024;  
    private State state = new State();  
    private BlockingCollection<string> outboundQueue = new  
BlockingCollection<string>();  
  
    private Action<RedisChannel, RedisValue> onRedisMessageHandler = null;  
    public Action<RedisChannel, RedisValue> OnRedisMessageHandler {  
        get {  
            if (this.onRedisMessageHandler == null) {  
                this.onRedisMessageHandler = new Action<RedisChannel,  
RedisValue>((channel, value) => this.outboundQueue.Add(value));  
            }  
            return this.onRedisMessageHandler;  
        }  
    }  
  
    //Constructor -receives the multiplexer  
    public ChatClient(ConnectionMultiplexer mux) {  
        this.state.subscriber = mux.GetSubscriber();  
    }  
}
```

```

        this.state.redisDB = mux.GetDatabase();
    }

    public async Task RunAsync(WebSocket socket) {
        this.state.outboundTask = Task.Run(async()=>await
OutboundLoopAsync(socket));
        await this.InboundLoopAsync(socket);
    }

    public async Task OutboundLoopAsync(WebSocket socket){
        foreach (var item in this.outboundQueue.GetConsumingEnumerable()) {
            var bytes = Encoding.UTF8.GetBytes(item);
            await
socket.SendAsync(bytes,WebSocketMessageType.Text,true,CancellationToken.None);
        }
    }

    private async Task InboundLoopAsync(WebSocket socket) {

        byte[] inboundBuffer = ArrayPool<byte>.Shared.Rent(BUFFER_SIZE);
        try {
            while (true) {
                WebSocketReceiveResult wsResult = await
socket.ReceiveAsync(inboundBuffer,CancellationToken.None);
                if (wsResult.MessageType == WebSocketMessageType.Close) {
                    ArrayPool<byte>.Shared.Return(inboundBuffer);
                    return;
                }
                byte[] incomingBytes = inboundBuffer[0..wsResult.Count];
                WSMMessage message = JsonSerializer.Deserialize<WSMessage>
(Encoding.UTF8.GetString(incomingBytes));
                await this.HandleMessageAsync(message); //check next section !
            }
        } finally {
            await this.CleanupSessionAsync();
        }
    }

    // cleanup routine
    //cleans redis hashset containing subscribed channels && subscriptions to
said channels
    private async Task CleanupSessionAsync() {
        foreach (var channelHash in await
this.state.redisDB.HashGetAllAsync(this.state.ClientId)) {
            await
this.state.subscriber.UnsubscribeAsync(channelHash.Name.ToString()),
this.OnRedisMessageHandler);
        }
        await this.state.redisDB.KeyDeleteAsync(this.state.ClientId);
    }
}

```

The **RunAsync** is the entrypoint to our client ; it starts the asynchronous **outboundTask** and then continues with running the inbound task.

When the `inboundTask` is finished/throws exception , we run the `CleanupSessionAsync` which deletes channel subscriptions as well as client data stored in a redis hashset.

The `OnRedisMessageHandler` delegate adds new messages to the `outboundQueue` and is needed to be provided when subscribing/unsubscribing on a target channel.

The `outboundTask` is a long running loop , that pops messages off the `outboundQueue` or waits in case of none present (blocking operation) ; the popped message is then written over the websocket to the client.

**Note** : To better understand the scope of the two tasks check out the [Sequence Diagram](#) !

Now that all I/O operation logic has been written we are going to define the message types and their handlers:

## Message Types

```
public enum DISCRIMINATOR {
    CLIENT__SUBSCRIBE = 0,
    CLIENT_UNSUBSCRIBE = 1,
    CLIENT_MESSAGE = 3,
    CLIENT_GET_CHANNELS=4,
    SERVER__RESULT=100 //server response for a given client request
}
```

**Note** - All messages that travel over the websocket both inbound and outbound comply to the format of [WSMessage](#) .

## Dispatcher

Whenever a new inbound message arrives we call the below method.

Depending on the [WSMessage Kind](#) we will deserialize the [WSMessage Payload](#) in a [Control Message](#) or a [ChatMessage](#) .

```
private async Task HandleMessageAsync(WSMessage message) {
    switch (message.Kind) {

        case WSMessage.DISCIMINATOR.CLIENT__SUBSCRIBE:
            ControlMessage subscribeMessage =
                JsonSerializer.Deserialize<ControlMessage>(message.Payload);
            await this.HandleSubscribeAsync(subscribeMessage);
            break;
        case WSMessage.DISCIMINATOR.CLIENT_UNSUBSCRIBE:
            ControlMessage unsubscribeMessage =
                JsonSerializer.Deserialize<ControlMessage>(message.Payload);
            await this.HandleUnsubscribeAsync(unsubscribeMessage);
            break;
        case WSMessage.DISCIMINATOR.CLIENT_MESSAGE:
            ChatMessage chatMessage =
                JsonSerializer.Deserialize<ChatMessage>(message.Payload);
            await this.HandleMessageAsync(chatMessage);
            break;
        case WSMessage.DISCIMINATOR.CLIENT_GET_CHANNELS:
            await this.HandleGetChannelsAsync(message);
            break;
    }
}
```

Next we are going to see how each message is being treated. The source file can be found [here](#) as well .

## Susbscribe Handler

```
private async Task HandleSubscribeAsync(ControlMessage subscribeMessage) {
    WSMMessage outboundMessage = null;
    if (subscribeMessage.ClientId != this.state.ClientId &&
this.state.ClientId != null) {
        outboundMessage = new WSMMessage {
            Kind = WSMMessage.DISCriminator.SERVER__RESULT,
            Payload = $"Error: ClientId mismatch ! "
        };
        outboundQueue.Add(outboundMessage.ToJson());
        return;
    }
    if (await state.redisDB.HashExistsAsync(this.state.ClientId =
subscribeMessage.ClientId, subscribeMessage.Channel)) {
        outboundMessage= new WSMMessage {
            Kind = WSMMessage.DISCriminator.SERVER__RESULT,
            Payload = $"Error: ALREADY SUBSCRIBED TO CHANNEL
{subscribeMessage.Channel}"
        };
        outboundQueue.Add(outboundMessage.ToJson());
        return;
    }
    await this.state.subscriber.SubscribeAsync(subscribeMessage.Channel,
this.OnRedisMessageHandler);
    await state.redisDB.HashSetAsync(subscribeMessage.ClientId,
subscribeMessage.Channel, "set");
    outboundMessage = new WSMMessage {
        Kind = WSMMessage.DISCriminator.SERVER__RESULT,
        Payload = $"Subscribed to channel : {subscribeMessage.Channel}
SUCCESSFULLY !"
    };
    outboundQueue.Add(outboundMessage.ToJson());
}
```

In order for the client to subscribe to a channel he must provide a **ClientId** as well as a **ChannelId** .

For each client we store in redis an associated hashset of the form :

```
{ "client_x" , [ { "channel1":"set" } , {"channel2","set"} ,
{"channel3","set"} ] }
```

We will first test if the **ClientId** matches with the one in redis.If positive we will test if the target **Channel** is already present in the redis hashset.

If all is fine we will use the `SubscribeAsync` method for the target `Channel` providing the `OnRedisMessageHandler` as the required argument.

On any type of failure we will write in the `outboundQueue` a `SERVER_RESULT` type of message with the specific error message.

## Unsubscribe Handler

```
private async Task HandleUnsubscribeAsync(ControlMessage unsubscribeMessage) {
    WSMMessage outboundMessage = null;
    bool deleted = await state.redisDB.HashDeleteAsync(this.state.ClientId,
unsubscribeMessage.Channel);
    if (!deleted) {
        outboundMessage = new WSMMessage {Kind =
WSMessage.DISCIMINATOR.SERVER__RESULT, Payload = $" UNSUBSCRIBE UNSUCCESSFUL"};
        outboundQueue.Add(outboundMessage.ToJson());
        return;
    }
    await
this.state.subscriber.UnsubscribeAsync(unsubscribeMessage.Channel,
this.OnRedisMessageHandler);
    outboundMessage = new WSMMessage {
        Kind = WSMMessage.DISCIMINATOR.SERVER__RESULT,
        Payload = $" UNSUBSCRIBE SUCCESSFUL"
    };
    outboundQueue.Add(outboundMessage.ToJson());
}
```

We delete the hashset associated with the `ClientId` and then use `UnsubscribeAsync` providing the `OnRedisMessageHandler`.

The result of the operation is written to the `outboundQueue`.

## Chat Message Handler

```
private async Task HandleMessageAsync(ChatMessage chatMessage) {
    if (!await this.state.redisDB.HashExistsAsync(chatMessage.ClientId,
chatMessage.Channel)) {
        WSMMessage outboundMessage = new WSMMessage {
            Kind = WSMMessage.DISCIMINATOR.SERVER__RESULT,
            Payload = $"Can not send message. Client :
{chatMessage.ClientId} " +
                $"does not exist or is not subscribed to channel :
{chatMessage.Channel}"
        };
        outboundQueue.Add(outboundMessage.ToJson());
    }
    await this.state.subscriber.PublishAsync(chatMessage.Channel, $"Channel
: {chatMessage.Channel}, Sender : {chatMessage.ClientId}, Message :
{chatMessage.Message}");
}
```

If the client is subscribed to the target channel ( the channel name exists in the redis hashset) we publish the message to redis.

Otherwise we write the failed attempt to the `outboundQueue`.

## Get Channels Handler

```
private async Task HandleGetChannelsAsync(WsMessage message) {  
    var channels = await  
this.state.redisDB.HashGetAllAsync(this.state.ClientId);  
    outboundQueue.Add(new WsMessage {  
        Kind = WsMessage.DISCriminator.SERVER__RESULT,  
        Payload = channels.ToJson()  
    }  
    .ToJson());  
}
```

We retrieve all key-values from the `ClientId` redis hashset and forward the result to the `outboundQueue`.

The dispatcher as well as the handlers can be found [here](#)

# Prerequisites

---

## NET 5.0

For this application we are using .NET 5.0 and you can download it from [here](#).

## Redis

### Installing

For this solution you will need to install Redis Server . You can download it from [here](#).

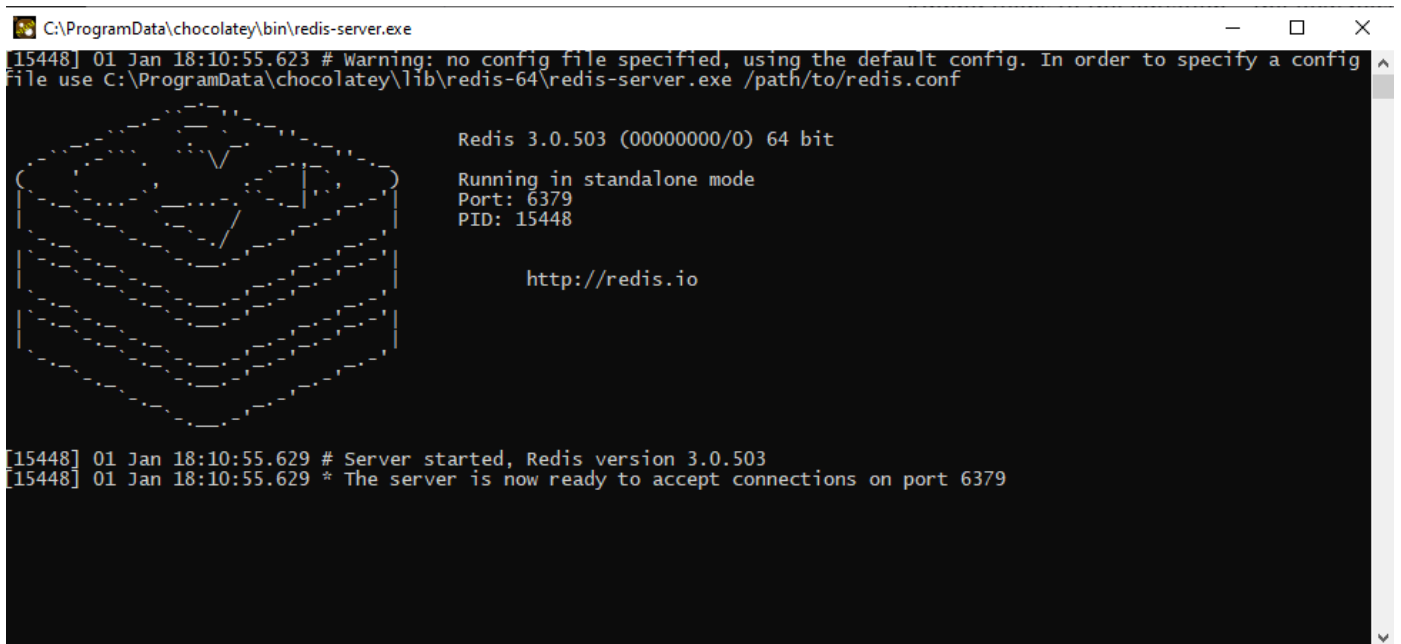
For windows users (me included) the easiest way to install redis is via the package manager *chocolatey* from [here](#) . Once installed from a terminal just run:

```
> choco redis
```

If the install was successful from a terminal run

```
> redis-server
```

and you should see the below output which indicates your redis server is up and running.



```
C:\ProgramData\chocolatey\bin\redis-server.exe
[15448] 01 Jan 18:10:55.623 # Warning: no config file specified, using the default config. In order to specify a config
file use C:\ProgramData\chocolatey\lib\redis-64\redis-server.exe /path/to/redis.conf

Redis 3.0.503 (00000000/0) 64 bit
Running in standalone mode
Port: 6379
PID: 15448

http://redis.io

[15448] 01 Jan 18:10:55.629 # Server started, Redis version 3.0.503
[15448] 01 Jan 18:10:55.629 * The server is now ready to accept connections on port 6379
```



## Redis-Cli

With the `redis-server` started you can start playing with redis using the `Redis-Cli` from a terminal with the command:

```
> redis-cli
```

Start having fun with redis by trying its [Commands](#) !

# Testing

---

I will be using [Simple WebSocket Client](#) as a testing interface.

## Subscribe

The screenshot shows the Simple WebSocket Client interface. At the top, the 'Server Location' section has a 'URL' field with 'ws://localhost:8600' and a 'Close' button. Below this, the 'Status' is 'OPENED'. The 'Request' section contains a text area with the JSON payload: `{"Kind": 0, "Payload": "{\"Channel\":\"mychannel\", \"ClientId\":\"adrian\"}"}`. Below the text area is a 'Send' button with the shortcut '[Shortcut] Ctr + Enter'. The 'Message Log' section at the bottom has a 'Clear' button and displays two messages: a red line for the outgoing request and a black line for the incoming response: `{"Kind":100,"Payload":"Subscribed to channel :mychannel SUCCESSFULLY !"}.`

```
Server Location
URL: ws://localhost:8600 Close
Status: OPENED

Request
{"Kind": 0, "Payload": "{\"Channel\":\"mychannel\", \"ClientId\":\"adrian\"}"}

Send [Shortcut] Ctr + Enter

Message Log Clear
{"Kind": 0, "Payload": "{\"Channel\":\"mychannel\", \"ClientId\":\"adrian\"}"}
{"Kind":100,"Payload":"Subscribed to channel :mychannel SUCCESSFULLY !"}.
```

In the picture above in the **Message Log**

- The red line(s) are messages we send to the server
- The black line(s) are server sent messages

## Unsubscribe

Server Location

URL: ws://localhost:8600

Close

Status: OPENED

Request

```
{ "Kind": 1, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\" }" }
```

Send

[Shortcut] Ctr + Enter

Message Log

Clear

```
{ "Kind": 0, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\" }" }
{ "Kind": 100, "Payload": "Subscribed to channel : mychannel SUCCESSFULLY !" }
{ "Kind": 1, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\" }" }
{ "Kind": 100, "Payload": " UNSUBSCRIBE SUCCESSFUL" }
```

# Message

Now that you are comfortable with the [Simple WebSocket Client](#) we can try sending messages to ourselves like below:

Message Log

Clear

```
{ "Kind": 0, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\" }" }
{ "Kind": 100, "Payload": "Subscribed to channel : mychannel SUCCESSFULLY !" }
{ "Kind": 3, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\", \"Message\": \" This is a message \" }" }
Channel : mychannel, Sender : adrian, Message : This is a message
{ "Kind": 3, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\", \"Message\": \" This is another message \" }" }
Channel : mychannel, Sender : adrian, Message : This is another message
{ "Kind": 3, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\", \"Message\": \" Last message !!!! \" }" }
Channel : mychannel, Sender : adrian, Message : Last message !!!!
{ "Kind": 1, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\", \"Message\": \" Last message !!!! \" }" }
{ "Kind": 100, "Payload": " UNSUBSCRIBE SUCCESSFUL" }
{ "Kind": 3, "Payload": "{ \"Channel\": \"mychannel\", \"ClientId\": \"adrian\", \"Message\": \" Can i still send ? \" }" }
{ "Kind": 100, "Payload": "Can not send message. Client : adrian does not exist or is not subscribed to channel : mychannel" }
```

We subscribe to channel **mychannel**, we send some messages and then unsubscribe. As expected the last message will not get published since we unsubscribed from the target channel.

## Note:

For debugging/diagnosing purposes you can open **Redis-Cli** and via the **Pub/Sub** functionality , subscribe to the target **Channel** and see what messages are flowing through it , or push messages directly and see if they arrive in your [Simple WebSocket Client](#) . (Make sure you send only serialized **WSMessage**'s).

# Further developments

---

In this article we have developed the server of a chat application.

We have tested the application by using a simple websocket client as an extension of google chrome but as you might have already guessed , for any non-trivial scenario in which this application is going to be used , we are going to need a dedicated Chat Client.

This is precisely what the next article will be about. Stay tuned !

\$