

# Contents

<b>Introduction</b>	<b>2</b>
<b>Classes and interfaces</b>	<b>2</b>
First application	2
Step 1. Connecting to VIT Recognition System	2
Step 2. Setting up the recognition	5
Step 3. Asynchronous recognition	8
Variant 1. Recognition with events	14
Variant 2. Recognition with tasks	15
Simple client example (full listing)	16
Simple client example with tasks (full listing)	17
<b>Frequently asked questions</b>	<b>18</b>
(about sufficient number of IRecognitionClient instances)	18
(about usage of more than one client)	19
(about requirements for GUIDs)	19
(about recognition settings parameters)	19
(about telling apart the IRecognitionRequestsSourceSettings instances)	20
(about sending multiple recognition requests)	20
(about open recognition requests limit)	20
(about recognition speed)	20
(about recognition requests locks)	20
<b>Sample GUI client</b>	<b>20</b>

## Introduction

**VitML.Recognition.Client.dll** is the .NET Framework assembly that allows .NET Framework applications to communicate with VIT Recognition System.

The assembly is delivered as **VitML.Recognition.Client.zip** archive. The archive contains the following folders:

- **Samples** — some simple samples of code for quick integration;
- **VitML.Recognition.Client** — the assembly and its dependencies;
- **VitML.Recognition.Client.TestClient** — sample client application (.NET, WPF);
- **VitML.Recognition.Client.TestClient\_src** — source files of the sample client application.

The current document contains two main sections:

- “Classes and interfaces” — will guide you through the VIT Recognition System API.
- “Sample GUI client” — will introduce you to a sample GUI application that demonstrates the operation of VIT Recognition System API.

## Classes and interfaces

### First application

In the **Samples** folder, you will find a **first\_API\_usage\_sample.cs** file. The file provides an example of a simple client code which communicates with VIT Recognition System. This code is also provided after the description of classes and interfaces defined in the assembly.

Alternatively you can check a **first\_API\_usage\_sample\_tasks.cs** file that contains similar integration code but with C# Tasks.

### Step 1. Connecting to VIT Recognition System

First of all, you need to connect your program to the VIT Recognition System. Create an instance of the **RecognitionClient** class:

```
namespace VitML.Recognition.Client
{
```

```

public class RecognitionClient : IRecognitionClient
{
    public RecognitionClient(IConnectionSettings aConnectionSettings);
}

```

The constructor of the **RecognitionClient** class requires an instance of object that implements **IConnectionSettings** interface:

```

namespace VitML.Recognition.Client.Base
{
    /// <summary>
    /// Settings for recognition database connection
    /// </summary>
    public interface IConnectionSettings
    {
        /// <summary>
        /// Recognition database host machine IP-address
        /// </summary>
        string Host { get; }

        /// <summary>
        /// Recognition database port
        /// </summary>
        int Port { get; }

        /// <summary>
        /// Recognition access username
        /// </summary>
        string User { get; }

        /// <summary>
        /// Recognition access password
        /// </summary>
        string Password { get; }

        /// <summary>
        /// Recognition database name
        /// </summary>
        string DataBaseName { get; }
    }
}

```

There is a default implementation of the **IConnectionSettings** interface called **ConnectionSettings** class that has default property values:

```
private const string HostDefaultValue = "127.0.0.1";
private const int PortDefaultValue = 5432;
private const string UserDefaultValue = "autocode";
private const string PasswordDefaultValue = "autocode";
private const string DatabasenameDefaultValue = "autocode";
```

The **IRecognitionClient** interface (of the **RecognitionClient** class) defines the public API of the VIT Recognition System:

```
namespace VitML.Recognition.Client.Base
{
    /// <summary>
    /// Represents a client for the Vit Recognition System
    /// </summary>
    public interface IRecognitionClient : IDisposable
    {
        /// <summary>
        /// Occurs when the recognition request is handled and the recognition request
        result is received
        /// </summary>
        event EventHandler<RecognitionRequestResultEventArgs> RecognitionRequestResult;

        /// <summary>
        /// Sets settings of a recognition requests source. If the recognition requests
        source does not exist, this method creates it.
        /// </summary>
        /// <param name="aRecognitionRequestsSourceSettings">Settings of a recognition
        requests source</param>
        /// <exception cref="VitML.Recognition.Client.RecognitionClientException">
        /// Thrown if an error occurs while updating recognition requests source
        settings
        /// </exception>
        void CreateOrUpdateRecognitionRequestsSource(IRecognitionRequestsSourceSettings
        aRecognitionRequestsSourceSettings);

        /// <summary>
        /// Sends a recognition request to the server.
        /// </summary>
        /// <param name="req">A recognition request</param>
        /// <exception cref="VitML.Recognition.Client.RecognitionClientException">
        /// Thrown if an error occurs while sending the recognition requests to the
        server
        /// </exception>
```

```

    /// <returns>The recognition request id</returns>
    Guid SendRecognitionRequest(IRecognitionRequest req);

    /// <summary>
    /// Sends a recognition request to the server.
    /// </summary>
    /// <param name="req">A recognition request</param>
    /// <exception cref="VitML.Recognition.Client.RecognitionClientException">
    /// Thrown if an error occurs while sending the recognition requests to the
server
    /// </exception>
    /// <returns>Recognition task</returns>
    Task<IRecognitionRequestResult> Recognize(IRecognitionRequest req);
}
}

```

Example (taken from `first_API_usage_sample.cs`):

```

//create default server connection settings
IConnectionSettings connectionSettings = new ConnectionSettings();
//create recognition client for the server
IRecognitionClient recognitionClient = new RecognitionClient(connectionSettings);

```

## Step 2. Setting up the recognition

Before sending the recognition requests, you have to specify the options of license plate recognition. This may be done with **CreateOrUpdateRecognitionRequestsSource** method of **RecognitionClient** class. It receives an instance of the object that implements an **IRecognitionRequestsSourceSettings** interface:

```

namespace VitML.Recognition.Client.Base
{
    /// <summary>
    /// Contains a set of settings for the recognition requests source.
    /// A recognition request source is the a notion that contains a set of recognition
settings and associates them with unique id.
    /// In most of cases the recognition request source is associated with the camera.
    /// If recognition settings is suitable for images from two or more cameras you can
consider all of the cameras as one recognition requests source.
    /// </summary>
    public interface IRecognitionRequestsSourceSettings
    {
        /// <summary>

```

```

    /// Gets the id of the recognition requests source.
    /// The user is responsible for the uniqueness of the id.
    /// </summary>
    Guid Id { get; }

    /// <summary>
    /// Gets the name of the recognition requests source.
    /// </summary>
    string Name { get; }

    /// <summary>
    /// Gets the minimum size (in pixels) of license plates images. If their sizes
    are less than this value, the system will ignore such license plates.
    /// </summary>
    uint LicensePlateSizeMin { get; }

    /// <summary>
    /// Gets the maximum size (in pixels) of license plates images. If their sizes
    exceed this value, the system will ignore such license plates.
    /// </summary>
    uint LicensePlateSizeMax { get; }

    /// <summary>
    /// Gets the maximum count of star symbols which is used in order to substitute
    unrecognized symbols. If the count of unrecognized symbols exceeds this value the system
    ignores the license plate.
    /// </summary>
    uint LicensePlateStarMaxCount { get; }

    /// <summary>
    /// Gets the minimum value of the recognition validity. If the recognition
    validity is less than this value the system ignores the license plate.
    /// </summary>
    uint LicensePlateValidityMin { get; }

    /// <summary>
    /// Gets the list of license plate templates which the system have to recognize.
    /// </summary>
    IEnumerable<ILicensePlateTemplate> LicensePlateTemplates { get; }

    /// <summary>
    /// Gets the list of vertexes of the area on the image where the system have to
    recognize.
    /// </summary>
    IEnumerable<IVertex> Vertexes { get; }
}
}

```

The default implementation of this interface is provided by the **RecognitionRequestsSourceSettings** class:

```
namespace VitML.Recognition.Client
{
    public class RecognitionRequestsSourceSettings : IRecognitionRequestsSourceSettings
    {
        public RecognitionRequestsSourceSettings(Guid aId, string aName);
    }
}
```

The **RecognitionRequestsSourceSettings** constructor requires two arguments: an ID and the name of the *recognition requests source*. The notion of the *recognition source* name allows to group the recognition settings into a set with the unique identifier. For example, there might be a set of settings specified for some **Camera** object which requests recognition information.

Example (taken from **first\_API\_usage\_sample.cs**):

```
// create id for a new RecognitionRequestsSource (set of recognition settings)
Guid recSourceGuid = Guid.NewGuid();
// create name for a new RecognitionRequestsSource (set of recognition settings)
const string recSourceName = "Camera_X";
// create a IRecognitionRequestsSourceSettings instance
var recSourceSettings =
    new RecognitionRequestsSourceSettings(recSourceGuid, recSourceName);
// add new RecognitionRequestsSource (set of recognition settings) to the server
recognitionClient.CreateOrUpdateRecognitionRequestsSource(recSourceSettings);
```

Also, you can speed up the plate recognition by excluding areas that are certain to not contain an image of a license plate. To do this, use the **Vertexes** property of the **RecognitionRequestsSourceSettings** class. This property allows you to set a list of polygon vertices that describes an area of interest. The coordinates of a vertex are counted from the top left corner of a frame. Each vertex can be represented by an instance of the object that implements an **IVertex** interface:

```
namespace VitML.Recognition.Client.Base
{
    public interface IVertex
    {
        float X { get; }
        float Y { get; }
    }
}
```

```
}

```

The default implementation of this interface is the **Vertex** class:

```
namespace VitML.Recognition.Client
{
    public class Vertex : IVertex
    {
        public Vertex(float aX, float aY);
    }
}

```

Be careful, as **CreateOrUpdateRecognitionRequestsSource** method may throw an **RecognitionClientException** in a case of an error:

```
namespace VitML.Recognition.Client
{
    public class RecognitionClientException : Exception
    {
        public int ErrorCode { get; private set; }

        public RecognitionClientException(int aErrorCode, string aMessage, Exception aInnerException);
    }
}

```

### Step 3. Asynchronous recognition

To perform the recognition action you should prepare an instance of object, that implements **IRecognitionRequest** interface and holds the recognition request data and options:

```
namespace VitML.Recognition.Client.Base
{
    /// <summary>
    /// Represents a recognition request.
    /// </summary>
    public interface IRecognitionRequest
    {
        int Id { get; set; }

        Guid Guid { get; set; }
    }
}

```



```

    /// <summary>
    /// Gets the jpeg image that the system have to recognize.
    /// </summary>
    byte[] Image { get; }

    /// <summary>
    /// Gets the id of the recognition requests source.
    /// </summary>
    Guid RecognitionRequestsSourceId { get; }

    /// <summary>
    /// Gets the user's context. The context will include in the recognition request
result.
    /// </summary>
    string Context { get; }

    /// <summary>
    /// Gets the value that indicate whether include plate image to the result of
the request.
    /// </summary>
    bool ReceivePlateImage { get; }

    /// <summary>
    /// Gets the value that indicate whether include car image to the result of the
request.
    /// </summary>
    bool ReceiveBodyImage { get; }

    /// <summary>
    /// Gets the value that indicate whether include full image to the result of the
request.
    /// </summary>
    bool ReceiveFullImage { get; }

    /// <summary>
    /// Gets or sets request's timestamp.
    /// </summary>
    DateTime TimeStamp { get; set; }

    /// <summary>
    /// Create light copy of IRequest. Used for storing sent request in memory.
    /// </summary>
    /// <returns></returns>
    IRequest LightCopy();
}
}

```

The **IRecognitionRequest** interface is implemented by the **RecognitionRequest** class:

```
namespace VitML.Recognition.Client
{
    public class RecognitionRequest : IRecognitionRequest
    {
        public RecognitionRequest(Guid aRecognitionRequestsSourceId, byte[] aImage);
    }
}
```

The recognition result is described with **IRecognitionRequestResult** interface:

```
namespace VitML.Recognition.Client.Base
{
    /// <summary>
    /// Represents result for the recognition request
    /// </summary>
    public interface IRecognitionRequestResult
    {
        /// <summary>
        /// Gets the unique id of the license plate
        /// </summary>
        Guid Id { get; }

        /// <summary>
        /// Gets the status of the recognition request.
        /// </summary>
        Status Status { get; }

        /// <summary>
        /// Gets the recognition result information
        /// </summary>
        RecognitionResult RecognitionResult { get; }

        /// <summary>
        /// Gets the user's context that was passed by the user with the recognition
        request.
        /// </summary>
        string Context { get; }

        /// <summary>
        /// Gets the license plate image.
        /// </summary>
        byte[] PlateImage { get; }
    }
}
```

```

    /// <summary>
    /// Gets the car image.
    /// </summary>
    byte[] BodyImage { get; }

    /// <summary>
    /// Get the full image.
    /// </summary>
    byte[] FullImage { get; }
}
}

```

The **Status** enumeration is defined as follows:

```

namespace VitML.Recognition.Client
{
    public enum Status
    {
        Failed = 0,
        Timeout = 1,
        Succeed = 2
    }
}

```

**Status.Succeed** is returned when plate was recognized in request image.

**Status.Failed** is returned when there was no plate in request image.

**Status.Timeout** is returned when request is processing more than timeout time.

The **RecognitionResult** object has properties that contain all information about plate recognition.

The corresponding types are:

```

namespace VitML.Recognition.Client
{
    public class RecognitionResult
    {
        public DateTime BestTimestamp { get; private set; }
        public long Id { get; private set; }
        public Guid Guid { get; private set; }
        public int Lane { get; private set; }
        public VehicleInfo Vehicle { get; private set; }
        public int Image { get; private set; }
        public PlateInfo Plate { get; private set; }
        public PostInfo Post { get; private set; }
    }
}

```

```
public ChannelInfo Channel { get; private set; }

public RecognitionResult(
    string besttsfmt,
    long id,
    Guid guid,
    int lane,
    VehicleInfo vehicle,
    int image,
    PlateInfo plate,
    PostInfo post,
    ChannelInfo channel
);
}

public class VehicleInfo
{
    public string Category { get; private set; }
    public double Length { get; private set; }
    public double Weight { get; private set; }
    public double Width { get; private set; }

    public VehicleInfo(
        string category,
        double length,
        double weight,
        double width);
}

public class PlateInfo
{
    public string BackgroundColor { get; private set; }
    public string Color { get; private set; }
    public int CountryCode { get; private set; }
    public int CountyCode { get; private set; }
    public string Country { get; private set; }
    public int Template { get; private set; }
    public string Text { get; private set; }
    public int Validity { get; private set; }
    public PlateRectangle Rectangle { get; private set; }

    public PlateInfo(
        string bcolor,
        string color,
        int country,
        int county,
        string strcountry,
```

```

        int template,
        string text,
        int validity,
        PlateRectangle rect);
    }

    public class PlateRectangle
    {
        public int Bottom { get; private set; }
        public int Left { get; private set; }
        public int Right { get; private set; }
        public int Top { get; private set; }

        public PlateRectangle(int bottom, int left, int right, int top);
    }

    public class PostInfo
    {
        public Guid Guid { get; private set; }
        public long Id { get; private set; }
        public string Name { get; private set; }

        public PostInfo(Guid guid, long id, string name);
    }

    public class ChannelInfo
    {
        public string Description { get; private set; }
        public long Id { get; private set; }
        public string Name { get; private set; }

        public ChannelInfo(string description, long id, string name);
    }
}

```

There are 2 ways to perform plate recognition action:

1. With events.
2. With tasks.

### Variant 1. Recognition with events

As you have already created a *recognition requests source*, in order to receive a recognition request result, you should subscribe to the **RecognitionRequestResult** event of delegate type **EventHandler<RecognitionRequestResultEventArgs>**.

Example of subscription (taken from **first\_API\_usage\_sample.cs**):

```
// subscribe on the recognition result events
recognitionClient.RecognitionRequestResult += RecognitionRequestResultEventHandler;

//...
private static void RecognitionRequestResultEventHandler(object sender,
    RecognitionRequestResultEventArgs e)
{
    //...
}
```

The **sender** parameter is a reference to the object, that invoked the event.  
The type of the **e** parameter is a **RecognitionRequestResultEventArgs**:

```
namespace VitML.Recognition.Client
{
    /// <summary>
    /// Provides data for the VitML.Recognition.Client.RecognitionRequestResult event
    /// </summary>
    public class RecognitionRequestResultEventArgs : EventArgs
    {
        /// <summary>
        /// Gets the result for the recognition request
        /// </summary>
        public IRecognitionRequestResult RecognitionRequestResult { get; private set; }

        public RecognitionRequestResultEventArgs(IRecognitionRequestResult
aRecognitionRequestResult);
    }
}
```

Send recognition requests by calling the **SendRecognitionRequest** method. The method requires the **IRecognitionRequest** as an argument.

Example (taken from **first\_API\_usage\_sample.cs**):

```
// get image that we want to recognize
byte[] imageBytes = new byte[0];
// create recognition request
var recognitionRequest = new RecognitionRequest(recSourceGuid, imageBytes);
// send recognition request
recognitionClient.SendRecognitionRequest(recognitionRequest);
```

After the recognition completes on server, the recognition result will be available in a subscribed method as the **RecognitionRequestResult** property of the **RecognitionRequestResultEventArgs** instance (taken from **first\_API\_usage\_sample.cs**):

```
private static void RecognitionRequestResultEventHandler(object sender,
    RecognitionRequestResultEventArgs e)
{
    IRecognitionRequestResult recRequestResult = e.RecognitionRequestResult;
    Status status = recRequestResult.Status;
    if (Status.Succeed == status)
    {
        string plateText = recRequestResult.RecognitionResult.Plate.Text;
        string plateCountry = recRequestResult.RecognitionResult.Plate.Country;
        Console.WriteLine("Plate: '{0}' ({1})", plateText, plateCountry);
    }
}
```

## Variant 2. Recognition with tasks

The alternative way to perform recognition is to call **Recognize** method which returns an instance of **Task<IRecognitionRequestResult>**:

```
Task<IRecognitionRequestResult> Recognize(
    IRecognitionRequest recognitionRequest,
    CancellationToken cancellationToken = default(CancellationToken));
```

This method requires an **IRecognitionRequest** and the optional **CancellationToken** arguments. With the instance of **CancellationToken** you can cancel recognition task if it takes too long to complete.

Code sample (from **first\_API\_usage\_sample\_tasks.cs** file):

```
// async recognition with task
IRecognitionRequestResult recognitionRequestResult = await
recognitionClient.Recognize(recognitionRequest);
Status status = recognitionRequestResult.Status;
if (Status.Succeed == status)
{
    string plateText = recognitionRequestResult.RecognitionResult.Plate.Text;
    string plateCountry = recognitionRequestResult.RecognitionResult.Plate.Country;
    Console.WriteLine("Plate: '{0}' ({1})", plateText, plateCountry);
}
else if (Status.Failed == status)
{
    Console.WriteLine("Recognition failed");
}
```

## Simple client example (full listing)

```

using System;
using VitML.Recognition.Client;
using VitML.Recognition.Client.Base;

namespace ExampleClientApplication
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            // create default server connection settings
            var connectionSettings = new ConnectionSettings {Host = "127.0.0.1"};
            // create recognition client for the server
            var recognitionClient = new RecognitionClient(connectionSettings);
            // subscribe on the recognition result events
            recognitionClient.RecognitionRequestResult +=
                RecognitionRequestResultEventHandler;

            // create id for a new RecognitionRequestsSource (set of recognition
settings)
            Guid recSourceGuid = Guid.NewGuid();
            // create name for a new RecognitionRequestsSource (set of recognition
settings)
            const string recSourceName = "Camera_X";
            // create a IRecognitionRequestsSourceSettings instance
            var recSourceSettings =
                new RecognitionRequestsSourceSettings(recSourceGuid, recSourceName);
            // add new RecognitionRequestsSource (set of recognition settings) to the
server
            recognitionClient.CreateOrUpdateRecognitionRequestsSource(recSourceSettings);

            // get image that we want to recognize
            byte[] imageBytes = new byte[0];
            // create recognition request
            var recognitionRequest = new RecognitionRequest(recSourceGuid, imageBytes);
            // send recognition request
            recognitionClient.SendRecognitionRequest(recognitionRequest);
        }

        private static void RecognitionRequestResultEventHandler(object sender,

```



```

        RecognitionRequestResultEventArgs e)
    {
        IRecognitionRequestResult recRequestResult = e.RecognitionRequestResult;
        Status status = recRequestResult.Status;
        if (Status.Succeed == status)
        {
            string plateText = recRequestResult.RecognitionResult.Plate.Text;
            string plateCountry = recRequestResult.RecognitionResult.Plate.Country;
            Console.WriteLine("Plate: '{0}' ({1})", plateText, plateCountry);
        }
    }
}
}
}

```

## Simple client example with tasks (full listing)

```

using System;
using VitML.Recognition.Client;
using VitML.Recognition.Client.Base;

namespace ExampleClientApplication
{
    internal class Program
    {
        private static void Main(string[] args)
        {
            // create recognition client for the server
            var recognitionClient = new RecognitionClient(
                connectionSettings: new ConnectionSettings
                {
                    Host = "127.0.0.1"
                });

            // create a IRecognitionRequestsSourceSettings instance
            var recSourceSettings =
                new RecognitionRequestsSourceSettings(
                    aId: Guid.NewGuid(),
                    aName: "Camera_X");

            // add new RecognitionRequestsSource (set of recognition settings) to the
server
            recognitionClient.CreateOrUpdateRecognitionRequestsSource(recSourceSettings);

            // create recognition request
            var recognitionRequest = new RecognitionRequest(

```

```

        aRecognitionRequestsSourceId: recSourceSettings.Id,
        aImage: new byte[0]); // get image that we want to recognize

    // async recognition with task
    Task.Run(async () =>
    {
        IRecognitionRequestResult recognitionRequestResult = await
recognitionClient.Recognize(recognitionRequest);
        Status status = recognitionRequestResult.Status;
        if (Status.Succeed == status)
        {
            string plateText =
recognitionRequestResult.RecognitionResult.Plate.Text;
            string plateCountry =
recognitionRequestResult.RecognitionResult.Plate.Country;
            Console.WriteLine("Plate: '{0}' ({1})", plateText,
plateCountry);
        }
        else if (Status.Failed == status)
        {
            Console.WriteLine("Recognition failed");
        }
    });
}
}
}
}

```

## Frequently asked questions

(about sufficient number of `IRecognitionClient` instances)

**How many `IRecognitionClient` instances are required? One per server, one per camera, one per request?**

The client application requires at least one instance of `IRecognitionClient` per server.

(about usage of more than one client)

**Let's say I want to use multiple cameras. What is the right way to do it? I need as many connections/clients representing as many cameras as I have? Alternatively, can I use one connection, so I can differentiate somehow between cameras while sending the requests?**

You can use any number of clients. You can define a *post* for each camera by calling the `CreateOrUpdateRecognitionRequestsSource` method. When you receive a result you get a post GUID value from `RecognitionResult.Post.Guid` property value:

```

IRecognitionRequestResult recognitionRequestResult = e.RecognitionRequestResult;
Guid postGuid = recognitionRequestResult.RecognitionResult.Post.Guid;

```

So, you can map a post with a specific camera used by you. When you send a request by calling the **SendRecognitionRequest** method of the **IRecognitionClient** instance, you have to pass a post/camera GUID in order to determine for which post/camera the result comes (as in the example).

In order to avoid misunderstanding, please note that the *recognition results source* notion, which is used by the **CreateOrUpdateRecognitionRequestsSource** and **SendRecognitionRequest** methods, is a synonym for a post or a camera in the context that we have just considered.

(about requirements for GUIDs)

**Are there any requirements for GUIDs (IDs of the recognition request sources)?**

There are no special requirements for GUID values. In case of any issues (incorrect operation of your system), check the following:

- values defined by calling the **CreateOrUpdateRecognitionRequestsSource** method are correct;
- recognition settings that you have defined for the post are correct.

(about recognition settings parameters)

**What parameters do we exactly need for the settings? Can I manually do the set-up once (directly on the server), but use the settings later? In this case, I would be able to send the requests only, not the settings, so this would significantly speed up my work.**

All the needed settings are defined as properties of the **IRecognitionRequestsSourceSettings** class. Its constructor explicitly requires two values (see the definition), but other parameters have default values.

You have to send the settings only when you want to create a new set or update the already existing one.

(about telling apart the **IRecognitionRequestsSourceSettings** instances)

**How can I differentiate between various instances of **IRecognitionRequestsSourceSettings**?**

Client application assigns the unique ID to each **IRecognitionRequestsSourceSettings** instance.

(about sending multiple recognition requests)

**Do I have to call `CreateOrUpdateRecognitionRequestsSource` or can I just send in the images?**

You can perform **`SendRecognitionRequest`** call to the Vit Recognition Server only when you have registered a recognition post with **`CreateOrUpdateRecognitionRequestsSource`**. If the post is registered you can send multiple recognition requests to that post.

(about open recognition requests limit)

**Is there a limit of open requests?**

No. There is no limit for open recognition requests.

(about recognition speed)

**How often can I send the recognition requests?**

It is recommended to send requests not often than 1 frame per second. If you'll send requests more often than this, the server recognitions queue will become full and all next requests will have timeout recognition result.

(about recognition requests locks)

**Do I have to lock calls to `SendRecognitionRequest`?**

No. There are locks in all requests to `VitRecognitionServer` inside an assembly implementation.

## Sample GUI client

Application: **`VitML.Recognition.Client.TestClient.exe`**

Location (within the delivered package): **`VitML.Recognition.Client.TestClient`** folder

Usage instruction: <https://vitcompany.atlassian.net/wiki/pages/viewpage.action?pageId=42729475>

Usage example:

VIT Recognition Client View (SAMPLE)

**Server connection settings**

Host: 192.168.101.97  
Port: 5432  
User: autocode  
Password:  
Database: autocode


**Recognition settings**

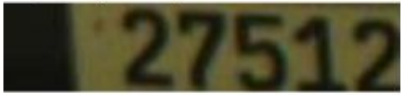
**Recognition request data**

Recognition requests source id: 0f8d4cfe-e4a5-4fc0-87a4-430a495e8c00  
Context:  
Receive plate image:   
Receive body image:   
Receive full image:   
Load image...  
Send recognition request

**Recognition response data**

Request id: 636f6a4d-3e1d-452a-8427-068cca690b30  
Status: Succeed  
Context:  
BestTimestamp = "11/28/2016 10:12:19"  
Id = "1480320739059504"  
Guid = "7f7e2f02-b542-11e6-a2da-6fbd5fcd10f8"  
Lane = "0"

Settings and request image:  


Response plate image:  


Response body image:  
Response full image:

VIT Recognition Client View (SAMPLE)


RecognitionResult

```

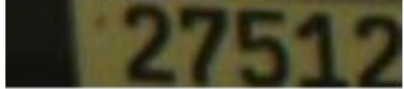
BestTimestamp = "11/28/2016 10:12:19"
Id = "1480320739059504"
Guid = "7f7e2f02-b542-11e6-a2da-6fbd5fcd10f8"
Lane = "0"
Vehicle.Category = "Unknown"
Vehicle.Length = "0"
Vehicle.Weight = "0"
Vehicle.Width = "0"
Image = "59492"
Plate.BackgroundColor = ""
Plate.Color = ""
Plate.CountryCode = "826"
Plate.CountyCode = "0"
Plate.Country = "United Kingdom"
Plate.Template = "0"
Plate.Text = "M2752"
Plate.Validity = "100"
Plate.Rectangle.Bottom = "509"
Plate.Rectangle.Left = "654"
Plate.Rectangle.Right = "870"
Plate.Rectangle.Top = "457"
Post.Guid = "33ce9c7c-a746-11e6-8f1f-1f0822a04da2"
Post.Id = "9199177"
Post.Name = "RecTestXX"
Channel.Description = ""
Channel.Id = "0"
Channel.Name = "0"

```

Settings and request image:



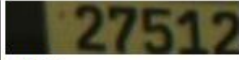
Response plate image:



Response body image:

Response full image:

Recognition history

Request Id	Status	Plate
636f6a4d-3e1d-452a-8427-068cca690b30	Succeed	 M2752