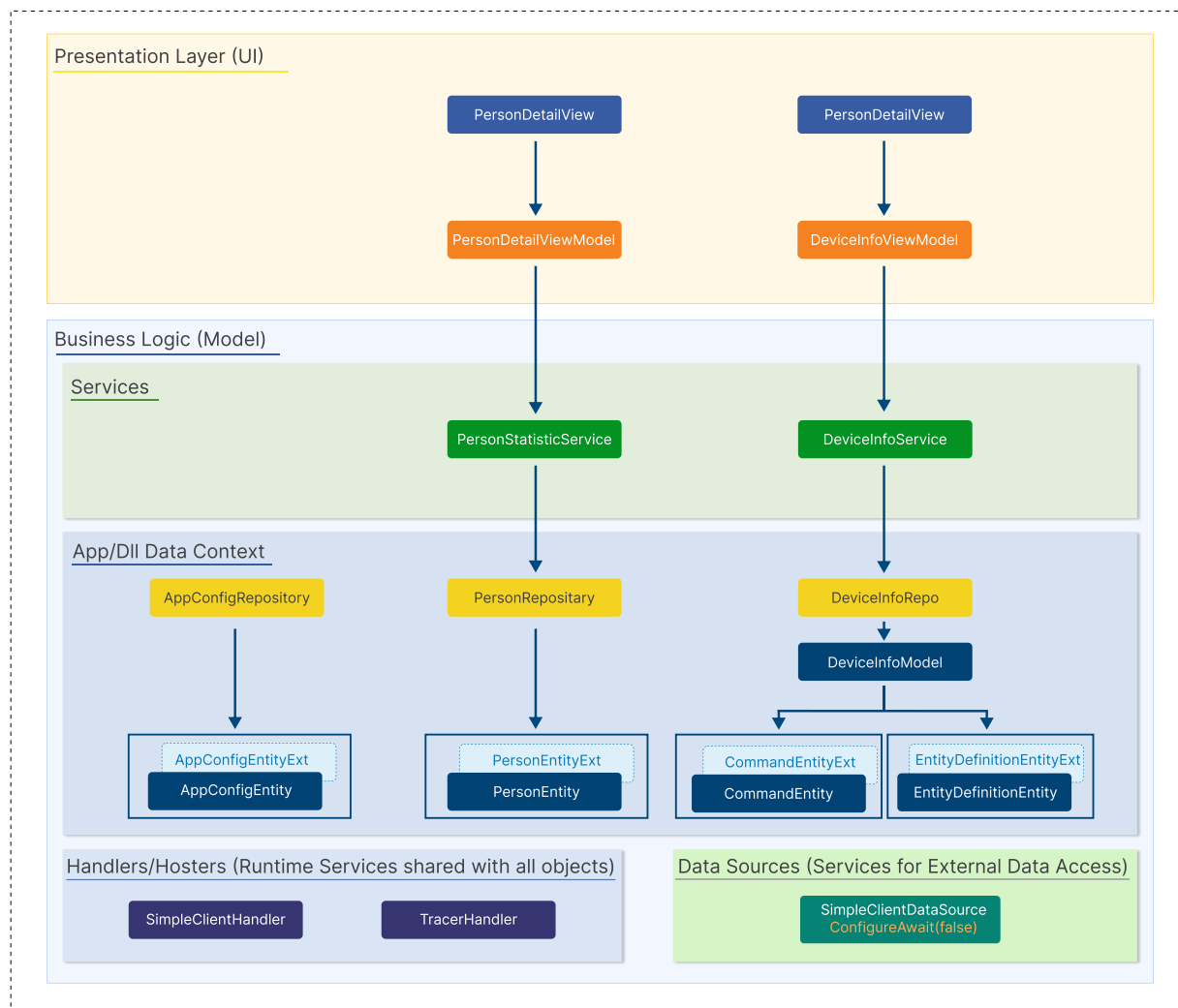


Gamanet Coding Convention

This document describes the specification of the application code structure applied to the development of any module or application in Gamanet. These coding and naming convention rules are mandatory for Gamanet programmers.

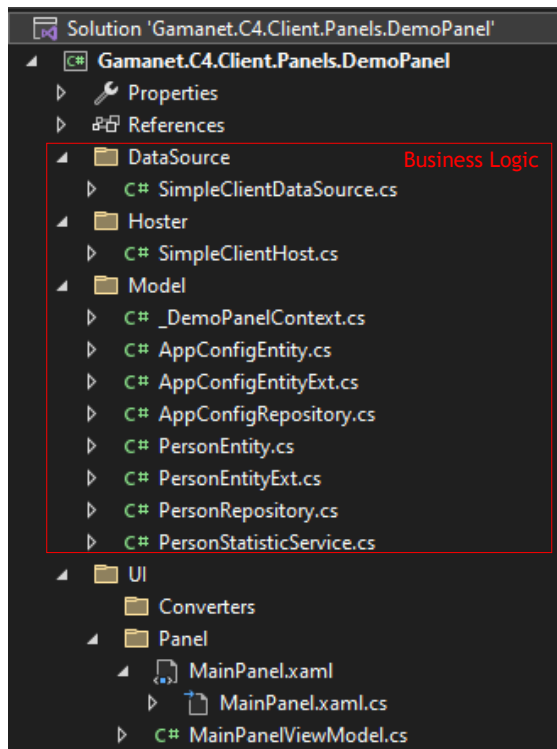
Any application code (exe) or module code (dll) can be viewed from several perspectives. For example, we can look at it from the point of view of its lifecycle, i.e., when the code is activated, when it is used, and when it is subsequently deactivated. That is, when a programming object (class) is created and when it is destroyed. We can also look at the code from the perspective of its use, whether it is a code that defines the structure of the data, or a code that subsequently processes the data.

The code structure within a module is divided into two parts. A standard client module has its presentation layer (User Interface), and the background logic (Business Logic). Business Logic processes the data entered by the user and ensures that it is retrieved or stored from/to an external source (server). Server modules, such as Drivers, IS Connectors or Smart Routines, have only a Business Logic layer and no UI part. Business Logic is further divided into several separate layers. Each layer has its own specific use case and its own lifecycle.



In a project, individual code parts are managed in a directory structure based on the category.

Within a project, the Business Logic is divided into directories by category. Entities and their associated Repositories and Services are located in the Model directory. Support Services such as Hosters and DataSource have their own folders. The presentation layer is located under the UI directory and is organized by object type (Window, Panel, Converter, Control, ...). The rule is that View and ModelView must always be stored next to each other. Thanks to the Naming convention, they are always sorted next to each other.

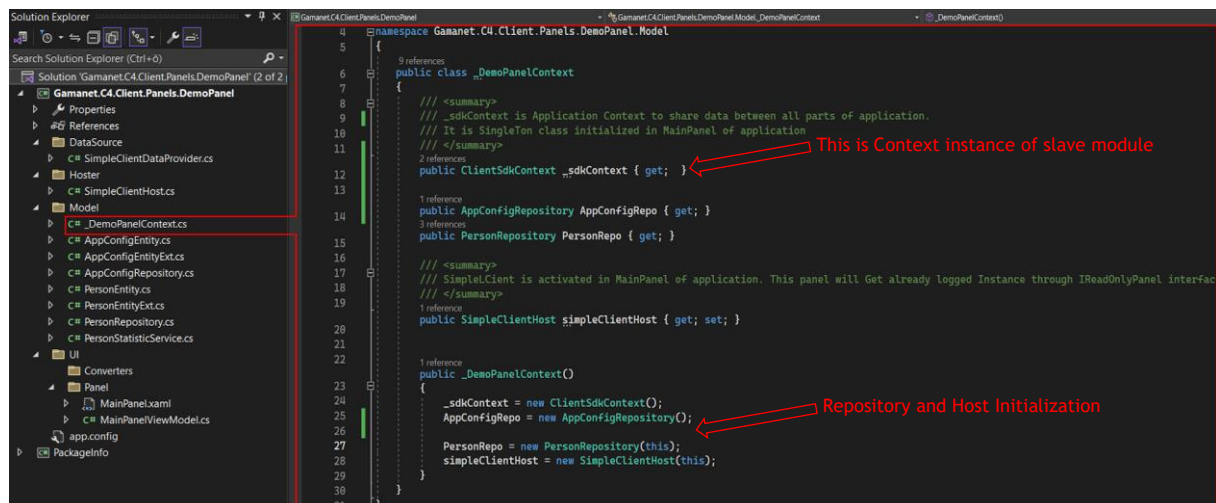


What is a Context object and what is it used for?

Each module, within its business logic, manages certain data that must be available throughout the module's lifecycle and must be available to all parts of the code in the module. All data shared in this way is registered and distributed through the Context object. Each module has only one Context at any time.

Its naming is based on the prefix "_", the name of the module, followed by the suffix "Context". In the case of the final application, it is an Application Context whose name is always _AppContext. In the case of a DLL, it is a Module Context, where the name is formed by the prefix "_", followed by the module name and the suffix "Context". If the module under development uses child modules that have their own Context, these are registered as instances in the Context of the module under development. The name of the Context instance is always in the form of the Context name in the Camel naming convention, possibly by shortening the module name to Initials. For example, if it is an _AppContext, its instance name will be defined as _appContext. If it is a _DemoPanelContext, the instance name is in the form _dpContext.

The Context is then distributed as a mandatory parameter in the constructor of each code object within the module's business logic. Exceptions are objects of type Entity and EntityExt, which do not have a defined constructor.



What is an object of type Entity

An Entity is an object that defines the data structure for a particular logical unit. These types of objects have the suffix Entity in their naming convention. They exclusively contain only properties of individual data types.

Within each module, there can be one special Entity of type AppConfigEntity. We will describe the specifics of this Entity at the end of this chapter.



Each Entity object must be inherited from the `PropertyChangedBase` object, which is located in the `Gamanet.Common` library. `Gamanet.Common` is a library of basic functions that is used in most applications developed by Gamanet. Nuget is available on Gamanet's public Nuget Servers.

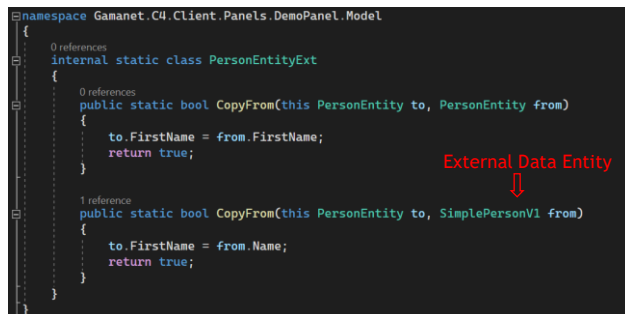
An Entity must be defined in the following structure. Each property must contain a private field that begins with an underscore, and that must be initialized to a default value. In this case, it is a string with the default value `String.Empty`. Furthermore, it must contain a public property of type getter and setter. Within the setter, the Property must have notification of changes implemented. This notification is only invoked if a new value has actually been assigned to the property. This check is a very important part of the whole code model. In the case of complex property types of type `Collection` or `Object`, the check on `Equals` is omitted. Thus, in the case of a complex object of type Entity, such a check is performed on its own properties.

Important Notice:

To ensure correct bindings, complex properties are never split. Instead, they are cleared and repopulated via the `EntityExt` type object.

What is an object of type EntityExt

For each basic object of type Entity, an additional object of type EntityExt is always created. These objects are created as a static extension for the Entity. Its mission is to ensure that data is copied within objects of the same type. It also serves as a place to copy data from external objects. This task is provided by the CopyFrom function with a parameter of the type of the entity from which the copying takes place.



```
namespace Gamanet.C4.Client.Panels.DemoPanel.Model
{
    0 references
    internal static class PersonEntityExt
    {
        0 references
        public static bool CopyFrom(this PersonEntity to, PersonEntity from)
        {
            to.FirstName = from.FirstName;
            return true;
        }
        1 reference
        public static bool CopyFrom(this PersonEntity to, SimplePersonV1 from)
        {
            to.FirstName = from.Name;
            return true;
        }
    }
}
```

External Data Entity
↓

In the case of processing external data, such as SimpleClient data, the CopyFrom function is the only place in the code where data is exchanged between the internal Entity and the external data. Validation needs to be done within the CopyFrom function. If the external data does not match the expected structure or the values do not match the specification, an exception must be raised. This ensures that the module is checked for compatibility with the external data in a single layer of code. In the rest of the module, only custom Entities defined in the Model directory should be used.

In the case of a property type of a complex object, for example another Entity, the CopyFrom function of the child Entity is called in the CopyFrom function.

The name EntityExt is always derived from the name of the Entity with the suffix EntityExt. In the example, the PersonEntityExt object defines the operations performed on the PersonEntity object.

What is a Repository object?

A Repository is an object that holds an instance of the corresponding Entity. In the case of a list of Entities, these lists are always maintained in an object of type ObservableCollection to ensure notification of changes when bindings are made. The role of the Repository is to retrieve and store data from/to an external source via objects of type DataSource, or to provide basic CRUD operations for the Entity. It then provides this data to all consumers within the module.

Every single Repository must adopt the Context of the module in which it is created within its constructor. The only exception is a Repository of type AppConfigRepository, which itself is used to provide configuration data to other Repositories.

```
public class PersonRepository
{
    /// <summary>
    /// Context is injected always, doesn't matter if it is used or not.
    /// </summary>
    2 references
    private _DemoPanelContext _dpContext { get; }

    /// <summary>
    /// Bindable Property cannot be replaced with another. Always just Cleared and Filled with new data.
    /// </summary>
    4 references
    public ObservableCollection<PersonEntity> Persons { get; }

    1 reference
    public PersonRepository(_DemoPanelContext context)
    {
        _dpContext = context;
        Persons = new ObservableCollection<PersonEntity>();
    }

    /// <summary>
    /// LoadDataAsync is called from ViewModel. It is not called from constructor.
    /// </summary>
    /// <returns></returns>
    1 reference
    public async Task<bool> LoadDataAsync()
    {
        var persons = await new SimpleClientDataSource(_dpContext).LoadPersonsAsync();

        Persons.Clear();
        foreach (var item in persons)
            Persons.Add(item);

        return true;
    }
}
```

The name of the repository consists of the name of the Entity whose instances it manages, with the ending Repository. It is also located in the Model directory. In this example, it is a PersonRepository. The subsequent repository instance name is truncated to the Repo name to ensure that the Declarations and the object instance are distinguishable.

Important Notice:

As with objects of type EntityExt, lists of Entities (ObservableCollection<Entity>) are never deleted in Repository to ensure correct bindings. Instead, they are cleared and repopulated.

The Repository is a memory storage that maintains the data and performs only basic operation to retrieve it and maintain an up-to-date state of the data. All other operations on the data are performed by special objects of type Service

What is a Service object

All extension functions that use data from the Repository in some way are executed in Service objects. In the case of more complex modules, more such objects can be created. A Service object has a short life cycle. It is always created at the moment of use and immediately disappears when the corresponding function is called. Each Service must have the Context of the module as a parameter in the constructor. Thus, it has access to all Repositories with data within its module. It can then use them within the task it is to perform.

```
public class PersonStatisticService
{
    2 references
    private _DemoPanelContext _dpContext { get; }

    0 references
    public PersonStatisticService(_DemoPanelContext context)
    {
        _dpContext = context;
    }

    0 references
    public int GetPersonCountByName(string Name)
    {
        return _dpContext.PersonRepo.Persons.Count(x => x.FirstName.Equals(Name));
    }
}
```



Service has the same name as Entity and Repository, but the end suffix is Service. If we need more Services, we can insert, after the name of the Entity (Person) and the Suffix (Service), the Service characteristic. For example, in this case PersonStatisticService.

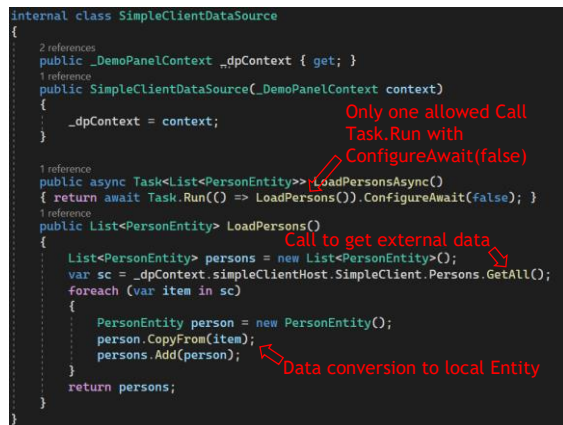
So we have defined Entity, EntityExt and Repository. The Entity defines the data structure. EntityExt provides for copying one Entity to another. The Repository contains instances of Entities which provides objects of type Service and ViewModel for processing

What are DataSource objects?

From a business logic perspective, there are several specific supporting object types. The first category is services that provide access to external data. External from the perspective of the module being developed. This category of objects is called DataSource services.

These are objects that provide for obtaining data or storing data from/to an external source. In the case of C4 Plugins development, these are data from C4 Server obtained through SimpleClient module. DataSource is the only object category that works with external Entity definitions. These are then converted into internal Entities used within the object using EntityExt objects. The return value from the DataSource functions must already be objects with Entities defined within the module.

This "two layers" concept ensures transparent debug of the entire application at the customer. If there is a problem in the availability of the application server, an exception will be raised in the DataSource module. If there is a problem in the correct data structure from the application server, an exception occurs in the EntityExt module in the CopyFrom function.



```
internal class SimpleClientDataSource
{
    2 references
    public _DemoPanelContext _dpContext { get; }
    1 reference
    public SimpleClientDataSource(_DemoPanelContext context)
    {
        _dpContext = context;
    }
    1 reference
    public async Task<List<PersonEntity>> LoadPersonsAsync()
    { return await Task.Run(() => LoadPersons()).ConfigureAwait(false); }
    1 reference
    public List<PersonEntity> LoadPersons()
    {
        List<PersonEntity> persons = new List<PersonEntity>();
        var sc = _dpContext.simpleClientHost.SimpleClient.Persons.GetAll();
        foreach (var item in sc)
        {
            PersonEntity person = new PersonEntity();
            person.CopyFrom(item);
            persons.Add(person);
        }
        return persons;
    }
}
```

Annotations in the image:

- Only one allowed Call Task.Run with ConfigureAwait(false) (pointing to the Task.Run call in LoadPersonsAsync)
- Call to get external data (pointing to the GetAll() call)
- Data conversion to local Entity (pointing to the CopyFrom call)

DataSource is also the only object where the use of Task objects for asynchronous data processing is allowed. An asynchronous function is always defined as an extension of a synchronous function. This will ensure easier calling of test functions within Unit Tests. Also, this is the only object where the use of function call.ConfigureAwait(false) is required when calling external functions with unknown behavior. This call ensures asynchronous processing of data from an external source outside of the MainThread application.

Warning:

DataSource is the only place to make this type of asynchronous call. Any other asynchronous call via Task.Run or .ConfigureAwait(false) requires Team Leader approval.

As Service category objects, their lifecycle is very short. Once the object is created and the corresponding function is called, they automatically expire. Object names contain the name of the data source with the DataSource ending. For more complex projects, multiple objects can be created with their mission specification in the name before the ending. For example, SimpleClientPersonsDataSource.

What are objects of type Handler/Host

In module development, there are always services that must run throughout the lifetime of a module. For example, a Simple Client instance is created and authorized, which still communicates with the server through the MessagePump submodule.

Such services that are used intermittently must be shared across the application, but are not direct data holders. The Simple Client itself does not hold any data, but we need its authorized instance to be available at all times for eventual retrieval of data via the SimpleClientDataSource modules. For this reason, they are created and shared within the Context module. By registering such Host/Handler modules within the Context module, they are then available to all services that use it

```
public class SimpleClientHost
{
    1 reference
    public _DemoPanelContext _dpContext { get; }
    2 reference
    public ISimpleClientV1 SimpleClient { get; set; }
    1 reference
    public SimpleClientHost(_DemoPanelContext context)
    {
        _dpContext = context;
    }

    0 references
    public void AttachSimpleClient(ISimpleClientV1 simpleClient)
    {
        SimpleClient = simpleClient;
        InitMessagePump();
    }

    1 reference
    private void InitMessagePump()
    {
        SimpleClient.MessagePump.OnMessage += MessagePump_OnMessage; ;
    }

    1 reference
    private void MessagePump_OnMessage(object sender, MessageEventArgsV1 e)
    {
    }
}
```

The objects are located in the Hoster directory. Depending on usage, their name is composed of the name of the object they manage and the Handler or Host ending, depending on the internal logic. The Host ending is used for objects that manage an instance of an external object - for example, SimpleClient. The Handler ending is used for objects that are themselves data handlers between module objects. For example, a TraceHandler is an object that provides central processing of error messages within a module.

How is configuration data managed?

There is a special Entity type within the entire module called AppConfig. It is the Entity that contains the configuration parameters needed to run the entire module.

```
7 references
public class AppConfigEntity : PropertyChangedBase
{
    /// <summary>
    /// This just for demo reason. In real world, you should use a real property.
    /// </summary>
    private string _connectionString = string.Empty;
    5 references
    public string ConnectionString
    {
        get => _connectionString;
        set { if(_connectionString != value) { _connectionString = value; OnPropertyChanged(); } }
    }
}
```

It is necessary to create an extension AppConfigEntityExt and also the corresponding Repository to AppConfigEntity. AppConfigEntityExt contains a special function CopySilentFrom, which ensures that data is loaded into the entity without notifying the rest of the module. This function is used when initializing configuration data or for modifying it by the user within the View.

```
0 references
internal static class AppConfigEntityExt
{
    0 references
    public static bool CopyFrom(this AppConfigEntity to, AppConfigEntity from)
    {
        to.ConnectionString = from.ConnectionString;
        return true;
    }

    0 references
    public static bool CopyFromSilent(this AppConfigEntity to, AppConfigEntity from)
    {
        bool ischange = false;
        if (to.ConnectionString.Equals(from.ConnectionString) == false)
        {
            ischange = true;
            to.SetFieldValue("_connectionString", from.ConnectionString);
        }

        return ischange;
    }

    1 reference
    public static void SetFieldValue(this AppConfigEntity obj, string name, object value)
    {
        var field = obj.GetType().GetField(name, BindingFlags.Public | BindingFlags.NonPublic | BindingFlags.Instance);
        field?.SetValue(obj, value);
    }
}
```

The AppConfigRepository object can retrieve configuration data by loading the app config, or by loading the ConnectionString from the App config and then retrieving the data from the Database.

The name of the AppConfigEntity instance in the AppConfigRepository is always called ConfigData.

```
public class AppConfigRepository
{
    1 reference
    public AppConfigEntity ConfigData { get; }
    1 reference
    public AppConfigRepository()
    {
        ConfigData = new AppConfigEntity();
    }

    0 references
    public bool LoadData()
    {
        return true;
    }

    0 references
    public bool SaveData()
    {
        return true;
    }
}
```

AppConfigRepository is the only Repository that has an empty constructor, that is, it does not need to be written. Other objects of type Repository, such as PersonRepository, must already take over in the Repository Context module.

Warning:

This block is omitted by default in the case of C4 Plugins development. The configuration data for the Plugin is provided by SdkContex. However, if necessary, its use is enabled and the data can be loaded into AppConfigEntity via the external SimpleClient module.

Now we have the basic structure of the business logic for the module under development.

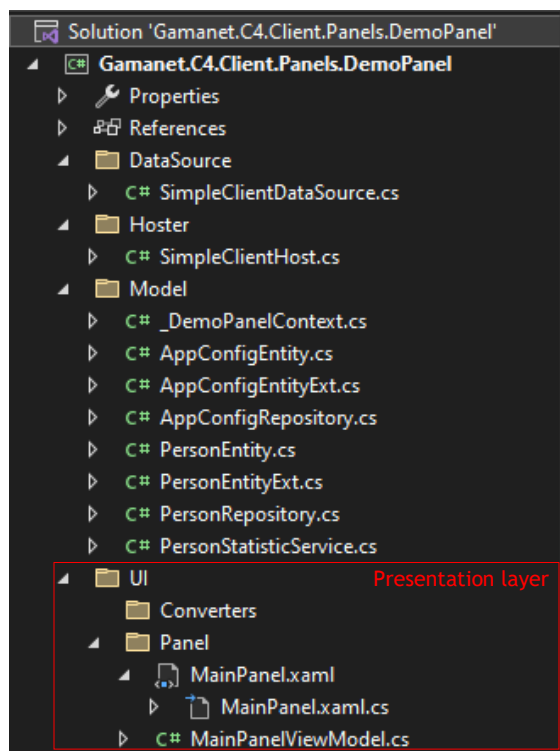
In the next section, we will show how to link the business logic to the user interface.

Presentation layer

The View and ViewModel codes programmed in Gamanet projects contain several specific rules that the developer must respect.

The presentation layer is always stored in the UI directory, which contains several subdirectories depending on the type of supporting objects, such as Converters or Panels. Panels then either contain all the panels in one directory or, in the case of a larger number, are distributed in subdirectories according to panels.

Each object of type View usually contains one object of type ViewModel. The names of the View and ViewModel objects must be the same and differ only in the ending. They must also be stored in the same directory so that they can be easily paired logically.



The rule that applies in Gamanet is that event handling from individual Controls placed in a View must be handled in functions within CodeBehind. It is strictly forbidden to use RelayCommand. Thus, CodeBehind usually serves only as a mapper between events from the View and function calls in the ViewModel.

The first child root UIElement (Grid) under UserControl/Page/Window within the View is used to attach the associated ViewModel. Therefore, it must be named "RootContainer", which will be referenced by the code in CodeBehind.

```

<UserControl x:Class="Gamanet.C4.Client.Panels.DemoPanel.UI.Panel.MainPanel"
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
xmlns:c4="http://schemas.c4portal.com/winfx/2023/presentation"
xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
xmlns:local="clr-namespace:Gamanet.C4.Client.Panels.DemoPanel.UI.Panel"
xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
d:DataContext="{d:DesignInstance Type=local:MainPanelViewModel}"
d:DesignHeight="450"
d:DesignWidth="800"
mc:Ignorable="d">
    <Grid x:Name="RootContainer">
        <c4:ToggleSingleButton Click="ToggleSingleButton_Click"
TextOff="Ola"
TextOn="Hi" />
    </Grid>
</UserControl>

```

Declaration of ViewModel for Intellisense

Container for ViewModel instance

Events from Controls handled in Code Behind

The method of linking a View to a ViewModel uses the View first rule. The Main Panel or Main Window initializes the overall Context of the module and maintains its instance throughout the lifetime of the module. In the case of the Main Window, the Context Module is attached as the main DataContext. This is then automatically inherited within the VisualTree to the child Controls.

Note:

In the case of panel development for a C4 application, the Context Module is only maintained as an instance in the Main Panel and is inherited programmatically towards child panels when needed.

The ViewModel is created as part of the View object initialization immediately after the DataContext is retrieved from the parent VisualTree for the base UI Element of the View object. This is usually an element of type UserControl or Page. Since the DataContext for the View is attached after the View object itself is created, we do not have a DataContext available in the View constructor. We only get it in the DataContextChanged event handler.

```

public partial class MainPanel : UserControl, IReadOnlyPanel
{
    3 references
    private _DemoPanelContext _dpContext { get; }
    2 references
    MainPanelViewModel _model { get; set; }
    0 references
    public MainPanel()
    {
        InitializeComponent();
        _dpContext = new _DemoPanelContext();
        this.DataContextChanged += MainPanel_DataContextChanged;
    }

    /// <summary>
    /// Just for case we want to use the parent data context for ViewModel
    /// </summary>
    /// <param name="sender"></param>
    /// <param name="e"></param>
    1 reference
    private void MainPanel_DataContextChanged(object sender, System.Windows.DependencyPropertyChangedEventArgs e)
    {
        RootContainer.DataContext = _model = new MainPanelViewModel(_dpContext);
    }

    0 references
    public void Initialize(ISimpleClient simpleClient)
    {
        // Here already logged instance of SimpleClient is injected to the context.
        _dpContext.SimpleClientHost.SimpleClient = (ISimpleClientVI)simpleClient;
    }

    1 reference
    private async void ToggleSingleButton_Click(object sender, System.Windows.RoutedEventArgs e)
    {
        await _model.LoadPersons();
    }
}

```

Module Context initialization in MainPanel

ViewModel Initialization with injection of Module Context

Events from Controls handled in Code Behind

The concept with RootContainer.DataContext is used to allow us to maintain our own DataContext in parallel and also the inherited DataContext that we got from the parent control. In the case of multiple UserControls that are nested inside each other, the Context of the entire module is usually obtained from the parent Control, and this is inserted as a parameter into the ViewModel constructor, and this ViewModel is attached as the data context of the RootContainer.

Subsequently, object instances are maintained in the ViewModel object for binding. Also, functions called from event handlers in CodeBehind are maintained in the ViewModel.

```
public class MainPanelViewModel
{
    private _DemoPanelContext _dpContext;

    /// <summary>
    /// This is way to create shortcut to objects managed by Context for simplification of code
    /// </summary>
    0 references
    private ClientSdkContext _sdkContext => _dpContext._sdkContext;

    1 reference
    public MainPanelViewModel(_DemoPanelContext context)
    {
        _dpContext = context;
    }

    1 reference
    public async void LoadPersons()
    {
        var ret = await _dpContext.PersonRepo.LoadDataAsync();
    }
}
```

Rules for coding modules and applications in Gamanet

In case of multithreaded communication and bulk data processing, always use Application.Current.Dispatcher.Invoke only once as the overall envelope for bulk processing.

The correct structure of a multithreaded call

```
Application.Current.Dispatcher.Invoke(() =>
{
    foreach()
    {
        Command;
    }
});
```

Forbidden multithreaded call structure!

```
foreach()
{
    Application.Current.Dispatcher.Invoke(() => Command);
}
```

