StarCommunity 3

A Programmer's Guide

StarCommunity -
Building Relationships

Netstar AB | Box 3415 | 103 68 Stockholm | Sweden
Phone: +46 (0)8 5000 98 70 | Fax. +46 (0)8 5000 98 71 | E-mail: products@netstar.se | Web site: www.netstar.se

Table of Contents

1. Basic Understanding

1.1. Getting Started

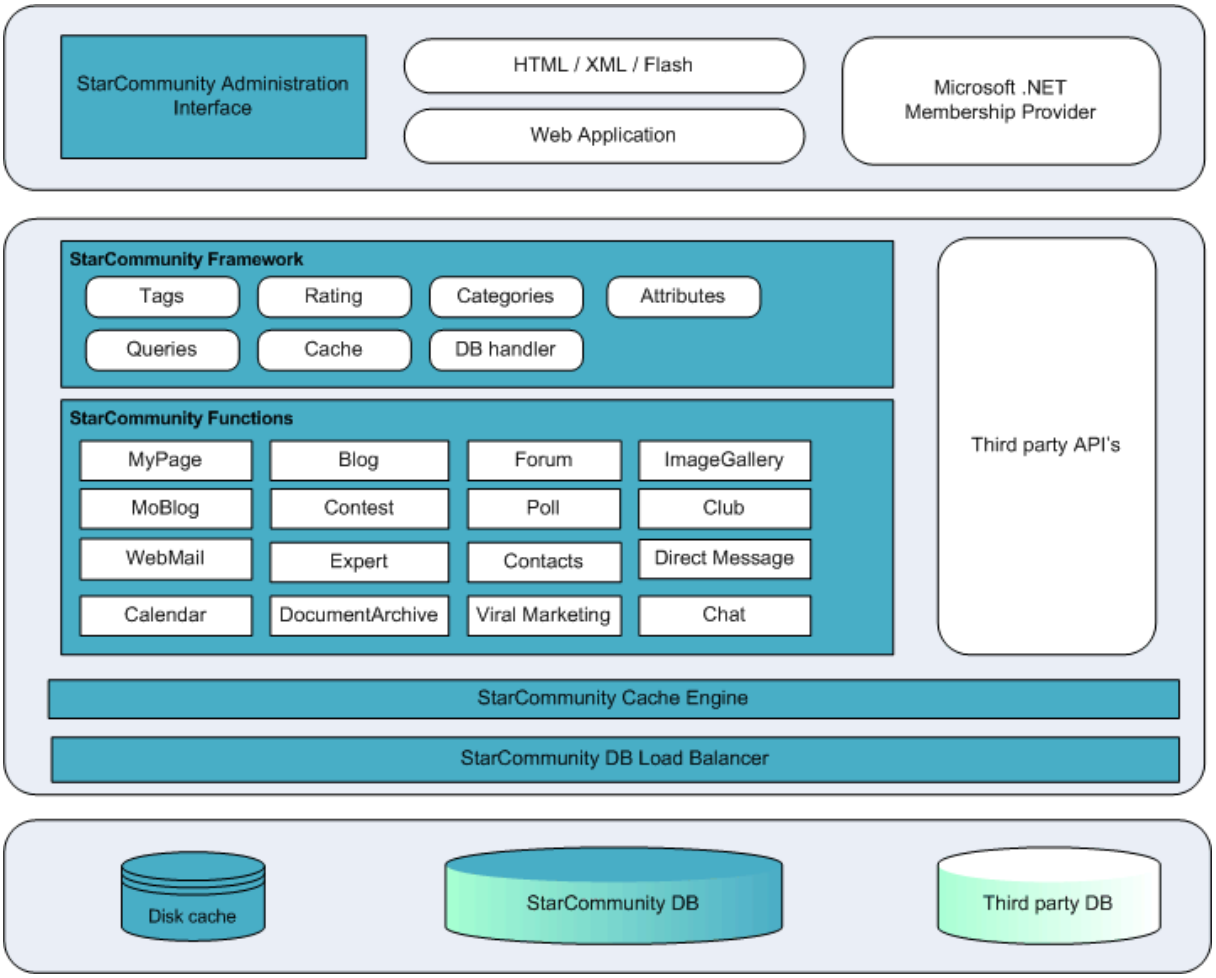
1.1.1. Setting up Visual Studio

After the installation of StarCommunity onto a Visual Studio Web Project the following steps needs to be taken.

- Add the necessary entries in the `Global.asax.cs` file mentioned in the Installation Manual under “EPiServer specific instructions”. This also applies to regular Web Projects not running EPiServer.
- Add all assemblies as a reference to the project. Especially in Visual Studio 2005 this is of great importance, since on a rebuild, Visual Studio will delete all unused assemblies from the `bin` directory.

1.1.2. What is the role of StarCommunity in an ASP.NET page?

StarCommunity is the backbone of a community. It is the API that retrieves and stores data using an object oriented structure and with high performance. The ASP.NET webpage comes into the picture when you want a way to display and input this data, which means, StarCommunity does not give you the set of web pages that makes up a community but it allows you to create them with full customizability in quicker and more stable way than you could do by coding a community from scratch.



1.2. Netstar Design Concept and Similarities

The Netstar framework design is written in such a way that developers will recognize the structure and immediately start development in new areas based on previous experience of StarCommunity development.

Classes that commit and retrieve data all end with “Handler”, e.g. `SiteHandler`.

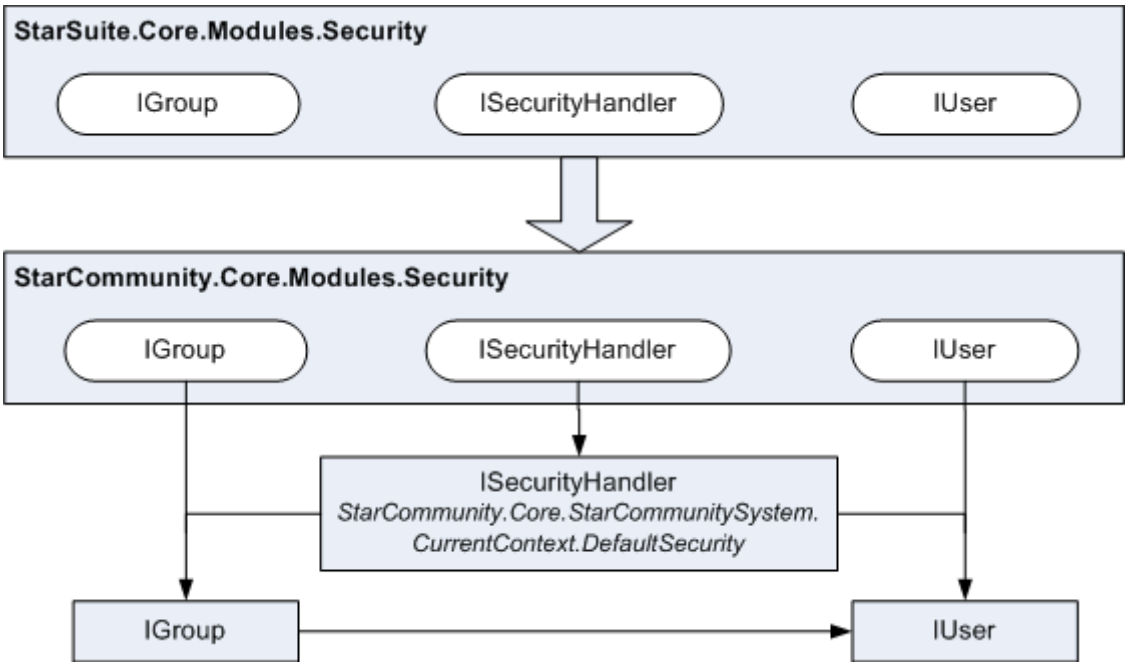
Committing data consists of methods starting with “Add”, “Update” and “Remove”.

Entity classes that hold data never contains methods for commiting data.

Handler classes contain events for most common methods, like adding, removing and updating data.

1.2.1.Required Framework Componenta

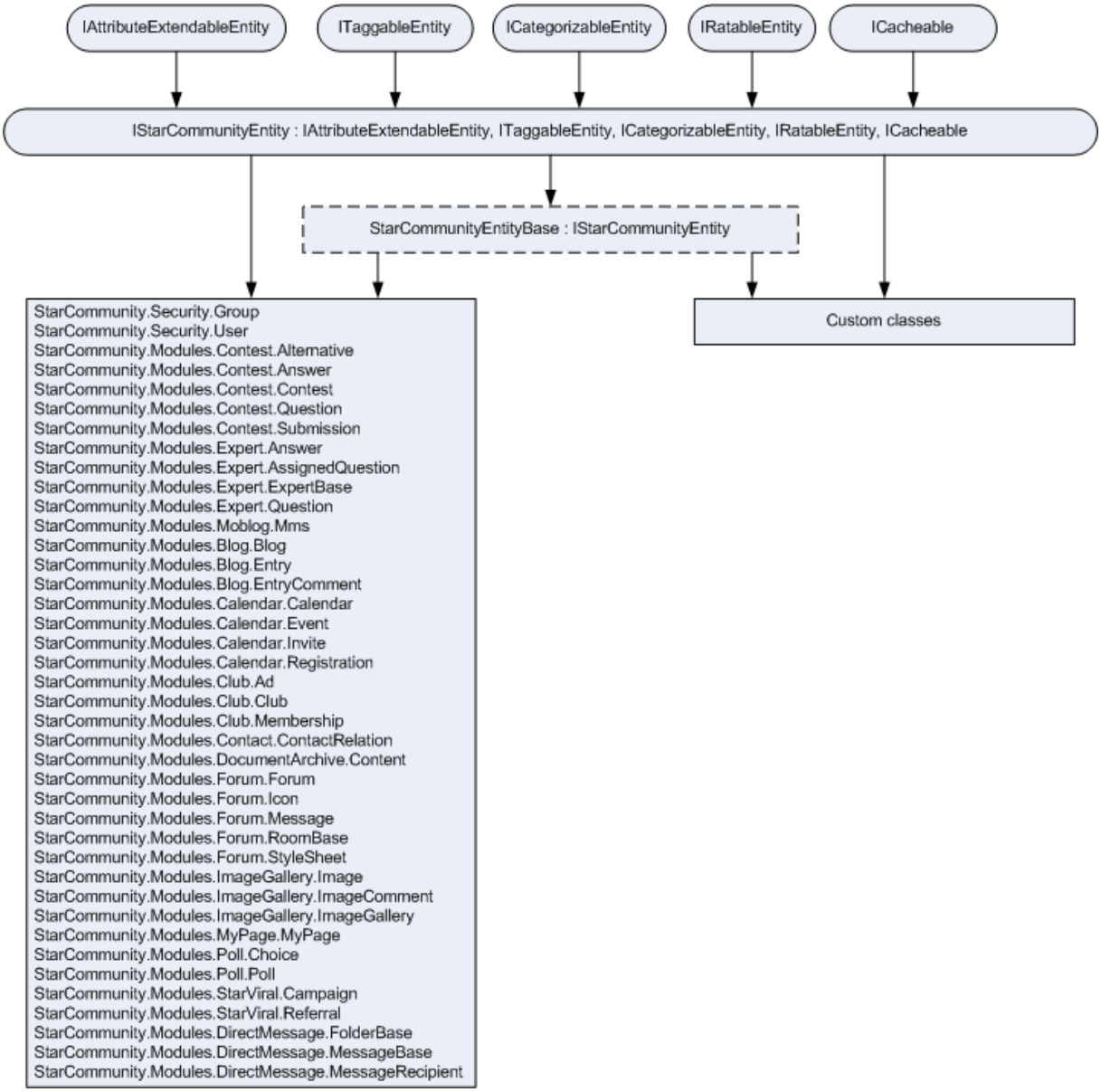
StarCommunity depends on a set of common classes, called “Required Framework Components” that reside in the `StarSuite`-namespace. These classes handle what is common between Netstar products, like site partitioning and security and access rights. The later is described in the figure below.



A Netstar product like StarCommunity is actually a module of Required Framework Components and when a web site is started it is these components that set up the necessary environment, loads the environment modules and provides a module context.

1.2.2.StarCommunity Entity Interfaces

The StarCommunity Entity Interfaces allows for developers to benefit from StarCommunity functionality such as Rating, Categorization, Tags, Cacheing, Attributes and Queries.

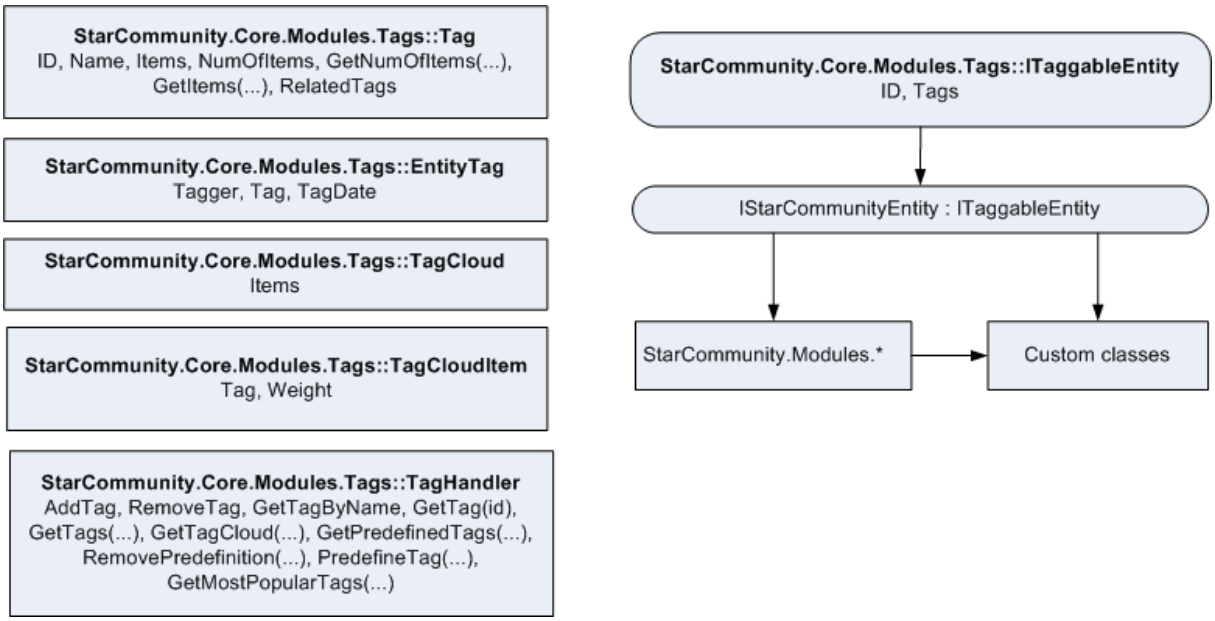


1.2.3.StarCommunity Core Modules

Included in the StarCommunity Core are several modules. The modules are Security, Tags, Rating, Categories, Attributes and Queries. These modules contains interfaces and classes that can be used throughout the StarCommunity system and also for third party classes that wish to benefit from this functionality.

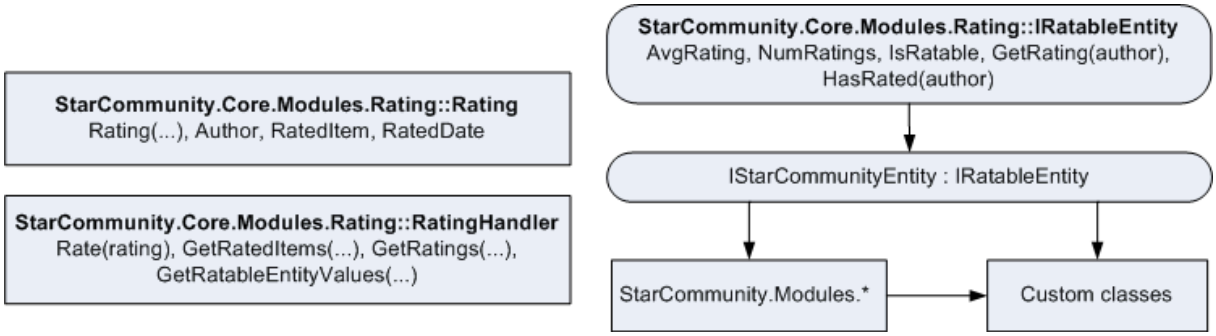
StarCommunity.Core.Modulse.Tags

Tags enable users to organize their content (often for the public) by tagging it with a certain word or phrase. All tags can then be merged into a Tag Cloud where tags are shown with different weight depending on the popularity. To use Tags, the class must implement the ITaggableEntity interface provided by StarCommunity Framework. The Tag system itself contains helper classes as shown in figure below. Coding samples using tags are found under the tutorial section in this document.



StarCommunity.Core.Modules.Rating

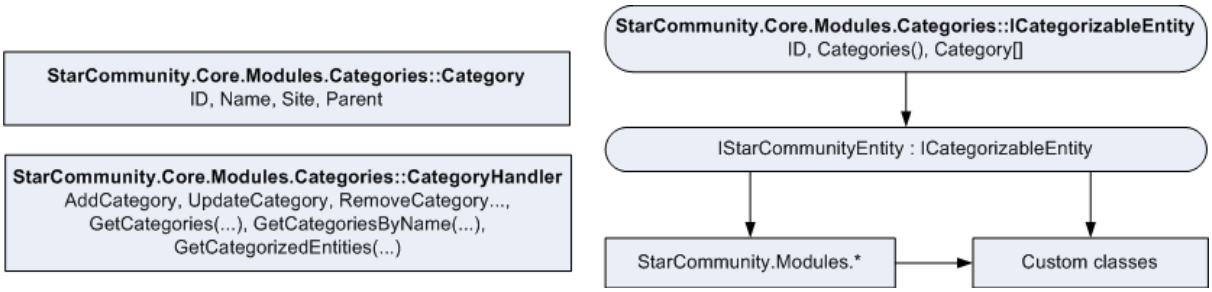
Rating enable developers to implement rating functionality for all classes that implements the IRatableEntity interface. Rated objects can be rated and retrieved based on their average rating. The rating system itself contains helper classes as shown in figure below. Coding samples using rating are found under the tutorials section in this document.



StarCommunity.Core.Modules.Categories

Categories enable developers to implement categorization functionality for all classes that implement the ICategorizableEntity. Interface. Categories can be either user defined or pre-defined and are stored in a tree structure. An Object can be categorized by binding

one or many categories to it and objects may then be retrieved based on their categories. Examples of content commonly categorized are Images, Blogs and Messages. The category system itself contains helper classes as shown in the figure below. Coding samples using Categories are found under the tutorial section in this document.



StarCommunity.Core.Modules.Attributes

Attributes enable developers to add custom attributes and attribute values of both primitive and complex types for all classes that implements `IAttributeExtendableEntity`interface. Attributes together with Queries makes StarCommunity a very flexible development platform, which allows system architects and developers to extend the core community functionality to meet highly specialized requirements.

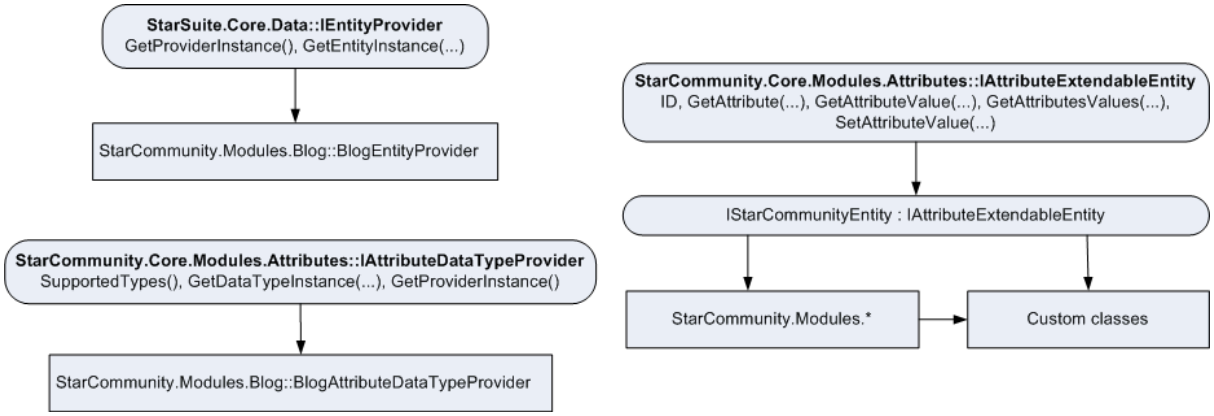
In its simplest form, attributes are used directly on existing StarCommunity objects in an ASP.NET page by using the Set/GetAttributeValue methods. The following code sets and gets an attribute named “forum_attribute” to the value of a forum instance for a `StarCommunity.Modules.ImageGallery.ImageGallery` object.

```
imageGallery.SetAttributeValue<Forum>(“forum_attribute", forum);
forum = imageGallery.GetAttributeValue<Forum>(“forum_attribute”);
```

In this approach the developer needs to keep track of the strings representing the attribute, which can be ok for systems with minor use of custom attributes. However, a more object-oriented approach is to create a new class that inherits `StarCommunity.Modules.ImageGallery.ImageGallery`with a fixed forum property.

The attribute system also enables the possibility to have third party classes as attributes to StarCommunity classes and have StarCommunity classes as attributes to third party classes.However, to use attributes in any other way than the simplest form described above, you need to define your own EntityProviders and AttributeDataTypeProviders to register new data types in the StarCommunity context. EntityProviders are described in section 1.2.4.

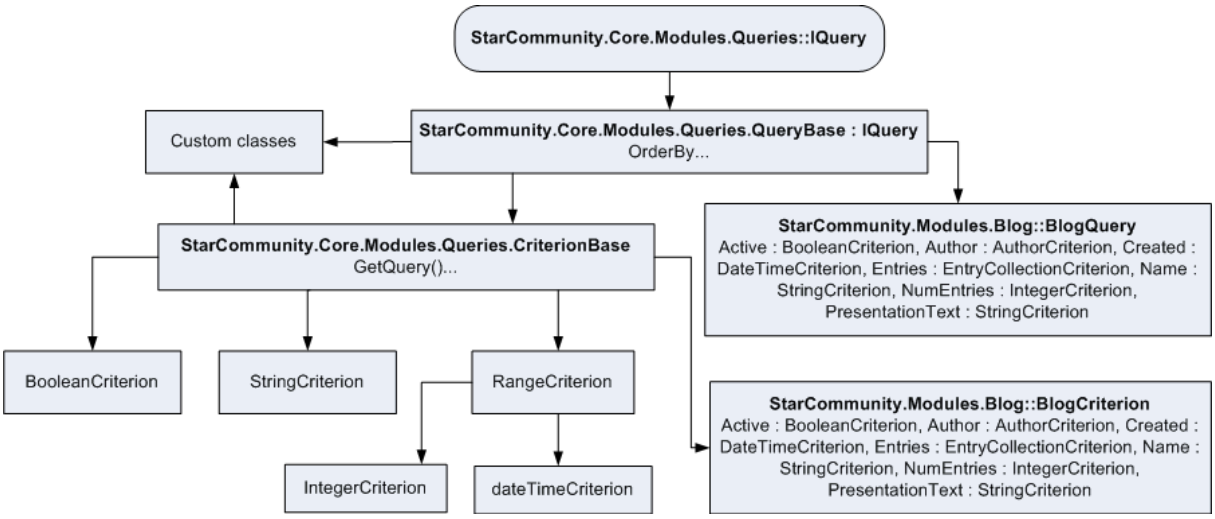
The attribute system itself contains helper classes as shown in the figure below. Coding samples using Attributes are found in the section 0.



StarCommunity.Core.Modules.Queries

Queries enables developers to get filtered collections of objects that have their database properties mapped in a configuration file and have a criterion class that is derived from CriterionBase and where the criterion class defines the filterable fields. All relevant StarCommunity classes are ready for filtering and retrieval. Coding samples using Queries are found in section 0.

The query system is described in the figure below using`StarCommunity.Modules.Blog`as an example.



1.2.4.StarCommunity EntityProviders

An `EntityProvider`is a singleton class that implements`StarSuite.Core.Data.IEntityProvider`and is responsible for returning instances of a specific type based on a `DbDataReader` or a unique ID. EntityProviders are already defined for all relevant StarCommunity classes.

Custom EntityProviders are necessary to define whenever third party classes are introduced in the StarCommunity context (For example custom classesthat inherits StarCommunity classes or implements StarCommunity interfaces). The following code snippets shows how to define a custom EntityProvider for the example custom class `MyImageGallery`that inherits `StarCommunity.Modules.ImageGallery.ImageGallery`.

```
public class MyImageGalleryEntityprovider : IEntityProvider
...
public object GetEntityInstance(Type type,  DbDataReader reader)
{
  if(type == typeof(ImageGallery) || type == typeOf(MyImageGallery)
return new MyImageGallery(reader);
}
...
public object GetEntityInstance(Type type, int Id)
{
if(type == typeof(ImageGallery) || type == typeOf(MyImageGallery)
return ImageGalleryHandler.GetImageGallery(id);
}
```

Now, we need to add the`Example.MyImageGalleryEntityProvider`to the EntityProvider.config file and thus override the `ImageGalleryEntityProvider`:

```
<EntityProvider>
<Name>Exempel.MyImageGalleryEntityProvider, Example</Name>
<SupportedTypes>
<SupportedType>
<Name>Example.MyImageGallery, Example</Name>
</SupportedType>
<SupportedType>
<Name>StarCommunity.Modules.ImageGallery.ImageGallery,
         StarCommunity.Modules.ImageGallery</Name>
</SupportedType>
</SupportedTypes>
</EntityProvider>
```

From now on, all requests for instances of the type `Example.MyImageGallery` and `StarCommunity.Modules.ImageGallery.ImageGallery` will be run through `Example.MyImageGalleryEntityProvider` and thus return an object of the `Example.MyImageGallery`type. Coding examples for creating entities are found in section 3.

1.2.5.StarCommunity AttributeDataTypeProviders

An `AttributeDataTypeProvider` is a singleton class that implements`StarCommunity.Core.Modules.Attributes.DataTypes.IAttributeDataTypeProvider`. An `AttributeDataTypeProvider`is responsible for returning `IAttributeDataType` instances made to handle a certain type. The `IAttributeDataType` should be able to translate this type into a primitive type that can be stored in the database.All relevant StarCommunity classes have AttributeDataTypeProviders defined. Custom AttributeDataTypeProviders are necessary to define whenever a custom datatypeis createdthat is used as a value for attributes.The following code snippets shows how to define a custom `AttributeDataTypeProvider` for the example custom class `MyClass`.

First we create the `AttributeDataTypeProvider` giving `MyClass` as a supported type. When `GetDataTypeInstance` is called with `MyClass` as the type argument, we take the primitive data type values and create an instance of our AttributeDataType class.

```
public class BlogAttributeDataTypeProvider
 : IAttributeDataTypeProvider
    {
        private static MyClassAttributeDataTypeProvider m_instance;

        private MyClassAttributeDataTypeProvider()
        {
        }

        #region IAttributeDataTypeProvider Members

        public Type[] SupportedTypes
        {
            get
            {
                return new Type[] { typeof(MyClass) };
            }
        }

        public IAttributeDataType
```

```
GetDataTypeInstance(TypecomplexType, List<object> dbValues)
        {
            if (complexType == typeof(MyClass))
                return new MyClassAttributeDataType(dbValues);
            else
                throw new NotSupportedException(String.Format("The
type '{0}' is not supported by this provider.",
complexType.ToString()));
        }

        public static IAttributeDataTypeProvider
GetProviderInstance()
        {
            if (m_instance == null)
                m_instance =
new MyClassAttributeDataTypeProvider();

            return m_instance;
        }

        #endregion
    }
```

If `MyClass` implements `IStarCommunityEntity` we can inherit`ComplexAttributeDataTypeBase<>`, which basically does all the work for us, we just define the complex datatype.

```
public class MyClassAttributeDataType :
ComplexAttributeDataTypeBase<MyClass>
{
public MyClassAttributeDataType(List<object> dbValues)
: base(dbValues, null, null)
    {
    }
}
```

If MyClass does not implement `IStarCommunityEntity` we inherit `AttributeDataTypeBase`, and we will have to do the conversion process on our own.

```
public class MyClassAttributeDataType : AttributeDataTypeBase
{
public MyClassAttributeDataType(List<object> dbValues)
            : base(dbValues, typeof(MyClass), typeof(Int32), null,
null)    {
    }

public override List<object> Values
        {
            get
            {
                List<object>objs = new List<object>();
                foreach (int id in DbValues)
                {
MyClass mc = MyClassHandler.GetMyClass(id);

                    if (mc != null)
objs.Add(mc);
                }
                return objs;
            }

            set
            {
                if (value == null)
                    throw new ArgumentNullException("value");

                List<object> dbValues = new List<object>();

                foreach (MyClassmc in value)
                    dbValues.Add(mc.ID);

                DbValues = dbValues;
            }
        }
}
```

Now, when we update an `IAttributeExtendableEntity` with an attribute of type `MyClass`, the `MyClassAttributeDataType` class will do the conversion.

1.1. Namespaces

1.2.6.StarSuite.Core

The `StarSuite.Core` namespace contains important startup classes like Settings, Site and SiteHandler and takes care of loading modules in the correct order based on dependencies.

1.2.7.StarSuite.Core.Cache

The `StarSuite.Core.Cache` namespace contains the Netstar Cache system. The cache is based on a tree-structure with the ability to have dependencies between branches. Every cached object in products like StarCommunity implements the ICacheable interface, allowing an object to have a primary cache key. The cache then keeps track of changes to this cache key and released other caches that also contain this object. All these features in conjunction make the Netstar cache a lot more precise than in previous versions.

The new cache system also implements the policy of read-only objects in cache. This is a big change since previous versions, since now objects retrieved from methods needs to be cloned before any properties are updated. All StarCommunity entities have a Clone() method that will return a writable copy of the object.

1.2.8.StarSuite.Core.Data

The StarSuite.Core.Data namespace contains the database communication layer. It is called by all Factory classes to open connections and transactions and makes it possible to run several method calls within one transaction.

```
bool alreadyInTransaction = DatabaseHandler.InTransaction;

if(!alreadyInTransaction)
DatabaseHandler.BeginTransaction();

try
{
    // execute a series of methods,
    // they will all be in the same transaction
    AddUser();
    SetAccessRights();

    // we are only responsible for commiting the transaction
    // if we were the ones to start it
    if(!alreadyInTransaction)
        DatabaseHandler.Commit();
} catch
{
    if(!alreadyInTransaction)
        DatabaseHandler.Rollback();
    throw;
}
```

1.2.9.StarSuite.Core.Modules

The `StarSuite.Core.Modules` namespace contains the classes and interfaces necessary for creating modules. To give an example of a module StarCommunity is actually one of them. And further down the module tree StarCommunity has its own modules.

1.2.10. StarSuite.Core.Globalization

The `StarSuite.Core.Globalization` namespace contains logic for retrieving and storing globalized and localized text strings that can be used on a web site of different languages.

```
// Get the translated text for the currently set culture
string t = GlobalizationHandler.GetTranslation("translation_key");
```

1.2.11. StarSuite.Core.Modules.Security

The StarSuite.Core.Modules.Security namespace contains the interfaces for users, groups and access rights. The logic is then implemented in different assemblies depending on data source. The `StarSuite.Security.Internal.dll` assembly (shipped with the installation) is an implementation of `StarSuite.Core.Modules.Security` that uses the SQL Server database as a source. By creating your own `SecurityHandler` the choosen data source would be without limits.

1.2.12. StarCommunity.Core

The `StarCommunity.Core` namespace contains the important `StarCommunitySystem` and `StarCommunityContext` classes that gives developers access to the StarCommunity `SecurityHandler`.

```
// Get the sql connection string from the starcommunity context
string cs = StarCommunitySystem.CurrentContext.SqlConnectionString;
```

1.2.13. StarCommunity.Core.Modules

This namespace contains the interface `IStarCommunityEntity` and the abstract implementation class`StarCommunityEntityBase`. `IStarCommunityEntity` implements the blueprint for tagging, attributes, rating and categorization. Also the Author classes and interfaces are located here, allowing for guests and users to identify themselves when making posts.

1.2.14. StarCommunity.Core.Modules.Security

The `StarCommunity.Core.Modules.Security` namespace contains extended interfaces based on the`StarSuite.Core.Modules.Security`namespace. Extensions include imlementation of `IStarCommunityEntity` on users and groups and the ability to store users for later activation by e-mail etc. The assembly `StarCommunity.Security.Internal.dll` is an implementation of this that uses the SQL Server database as a data source.

```
// Get the currently logged in user. DefaultSecurity should be
// StarComunity.Security.Internal.SecurityHandler since no other
// handler is installed.
```

```
IUser u = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;
```

1.2.15. StarCommunity.Core.Modules.Tags

This namespace contains the Tags core module whose functionality spans over all the StarCommunity modules, and optionally it may extend to third party modules as well. It allows for tagging an entity of any type (implementing the `ITaggableEntity` interface) with a tag. A tag cloud may then be generated for the tags globally or by site and/or type
See the section 0 for implementation details.

1.2.16. StarCommunity.Core.Modules.Rating

This namespace contains the Rating core module whose functionality spans over all the StarCommunity modules, and optionally it may extend to third party modules as well. It allows for rating an entity of any type (implementing the `IRatableEntity` interface) providing a rating value. Entities may then be retrieved based on their average rating.
See section 2.3 for implementation details.

1.2.17. StarCommunity.Core.Modules.Categories

This namespace contains the Categories core module whose functionality spans over all the StarCommunity modules, and optionally it may extend to third party modules as well. It allows for categorizing an entity of any type (implementing the `ICategorizableEntity` interface) providing one or many categories. Entities may then be retrieved based on their categorization.See section 2.4 for implementation details.

1.2.18. StarCommunity.Core.Modules.Attributes

This namespace contains the Attributes core module whose functionality spans over all the StarCommunity modules, and optionally it may extend to third party modules as well. It allows for binding attribute values of primitive or complex types to an entity of any type (implementing the `IAttributeExtendableEntity` interface). See section 2.5 for implementation details.

1.2.19. StarCommunity.Core.Modules.Queries

This namespace contains the Queries core module whose functionality spans over all the StarCommunity modules, and optionally it may extend to third party modules as well. It exposes the base functionality of queries and criteria and is not used directly, but instead through implementations of these base classes. Queries allows for retrieving dynamically filtered results.

1.2.20. StarCommunity.Modules.Blog

The `StarCommunity.Modules.Blog` namespace contains classes for creating and managing blogs.

1.2.21. StarCommunity.Modules.Calendar

The `StarCommunity.Modules.Calendar` namespace contains classes for creating and managing calendars, events, event invites and event registrations.

1.2.22. StarCommunity.Modules.Chat

The `StarCommunity.Modules.Chat` namespace contains classes for creating and managing chat rooms, chat users and chat moderators.

1.2.23. StarCommunity.Modules.Club

The `StarCommunity.Modules.Club` namespace contains classes for creating and managing clubs, club members, club ads and club keywords.

1.2.24. StarCommunity.Modules.ConnectionLink

The `StarCommunity.Modules.ConnectionLink` namespace contains classes for retrieving the shortest path between two users with the use of a breadth-first algorithm.

1.2.25. StarCommunity.Modules.Contact

The `StarCommunity.Modules.Contact` namespace contains classes for managing one-way or two-way relations between users. Create relations immediately or let users approve them by the use of relations of the type “Request”.

1.2.26. StarCommunity.Modules.Contest

The `StarCommunity.Modules.Contest` namespace contains classes for managing contests with alternative and free-text questions.

1.2.27. StarCommunity.Modules.DirectMessage

The `StarCommunity.Modules.DirectMessage` namespace contains classes for sending and receiving direct-messages. Messages can be sent to multiple recipients at once and also be used in “System” mode, which allows you to send messages to a large number of users without performance drop.

1.2.28. StarCommunity.Modules.DocumentArchive

The `StarCommunity.Modules.DocumentArchive` namespace contains classes for storing documents and creating folder structures.

1.2.29. StarCommunity.Modules.Expert

The `StarCommunity.Modules.Expert` namespace contains classes for creating and managing experts, assign questions, approve answers and synchronize with forum rooms.

1.2.30. StarCommunity.Modules.Forum

The `StarCommunity.Modules.Forum` namespace contains classes for creating forums and moderate topics.

1.2.31. StarCommunity.Modules.ImageGallery

The `StarCommunity.Modules.ImageGallery` namespace contains classes for creating image galleries, generating thumbnails, cropping, resizing, promoting and voting for images.

1.2.32. StarCommunity.Modules.Moblog

The `StarCommunity.Modules.Moblog` namespace contains classes for receiving MMS messages sent from mobile phones. Moblog comes integrated with the mobile enabler Unwire but can easily be integrated with any other enabler.

1.2.33. StarCommunity.Modules.Moblog.ContentProviders.Unwire

The `StarCommunity.Modules.Moblog.ContentProviders.Unwire` namespace contains the classes of the Unwire mobile enabler.

1.2.34. StarCommunity.Modules.MyPage

The `StarCommunity.Modules.MyPage` namespace contains classes for presenting a user, block other users and easily reach other modules connected to a user.

1.2.35. StarCommunity.Modules.NML

The `StarCommunity.Modules.NML` namespace contains classes for rendering HTML content based on a dynamically defined set of tags and attributes.

1.2.36. StarCommunity.Modules.OnlineStatus

The `StarCommunity.Modules.OnlineStatus` namespace contains classes for monitoring if a user is online, when the user last logged in or who is online at the moment.

1.2.37. StarCommunity.Modules.Poll

The `StarCommunity.Modules.Poll` namespace contains classes for creating and managing voting polls.

1.2.38. StarCommunity.Modules.StarViral

The `StarCommunity.Modules.StarViral` namespace contains classes for creating and managing viral marketing campaigns, follow user's recruitments and select the best recruiter.

1.2.39. StarCommunity.Modules.Webmail

The `StarCommunity.Modules.Webmail` namespace contains classes for sending and receiving e-mails over IMAP to be presented on the website. Automatically synchronizes with the community user database.

2. Tutorials

2.1. User Management

User Management in StarCommunity is done through a singleton of the type `StarCommunity.Core.Modules.Security.ISecurityHandler` that is reached through the property `DefaultSecurity` at `StarCommunity.Core.StarCommunitySystem.CurrentContext.DefaultSecurity`.

1.2.40. Adding a User

Among the first things you build into a community is the possibility to register a membership and get a `User` object instance representing this community member.

This article shows you, the developer, how to typically proceed to create this functionality with the help of the StarCommunity Framework.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a user. The namespaces `StarCommunity.Core` and `StarCommunity.Core.Modules.Security` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Declaring a New User Object

We then create a new `User` object instance by calling the `NewUser` property. This property always returns a new `User` object instance and is handled by the running `SecurityHandler`.

Currently the user exists only in memory. Before committing the object, we will need to set a minimum list of properties, or the API will throw an exception when we try to commit it to the database.

```
//Add user
IUser newUser = (IUser)StarCommunitySystem.
                       CurrentContext.DefaultSecurity.NewUser;
newUser.Alias = "Jhonny";
newUser.GivenName = "John";
newUser.SurName = "Doe";
newUser.BirthDate = new DateTime(1975, 3, 5);
newUser.EMail = "john@doe.com";
newUser.PassWord = "secret";
newUser.UserName = "john";
newUser.Culture = System.Globalization.CultureInfo.CurrentUICulture;
```

In the above example we end with setting the culture. The culture will be used to define the user's language preference. It is used in the administration interface, but if the user is not

intended to be an administrator, this culture can be used for other purposes. You can read more about attributes under **Fel! Hittar inte referenskälla.**.

Committing the User Object to Database

Up until now the user has only existed in memory, to finalize the creation of the user we need to commit it to the database. We do this by calling the `AddUser` method of the currently running `SecurityHandler`. Returned is the added user, but with the new unique `ID` property set. This object can now be used as a user representation.

```
newUser = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.AddUser(newUser);
```

1.2.41. Authenticating a User

When you want to authenticate a login request by a member of a community, this can be done through running `SecurityHandler` singleton.

Import Necessary Namespaces

First import the necessary namespaces that will be used to authenticate a user. The namespace `StarCommunity.Core` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
```

Performing the Authentication

Authentication will require the username and password entered by the user. The call to `AuthenticateUser` will return if it was a success or not, with the additional `user` out variable. The `user` variable will be set to the `User` object instance found if authentication was successful.

```
StarSuite.Core.Modules.Security.IUser user = null;
bool isAuthenticated = StarCommunitySystem.CurrentContext.
      DefaultSecurity.AuthenticateUser("john", "secret", out user);
```

This should set `isAuthenticated` to `true` and the `user` variable to the instance of the user we added in 1.2.40.

Where is the Authentication Ticket?

One important thing to remember is that StarCommunity provides the means for authenticating but does not set an actual authentication ticket in the ASP.NET authentication framework. To finalize the authentication this will have to be done manually.

ASP.NET Membership Provider

Will be added in a later revision of the StarCommunity 3.0 beta and will be mentioned in this document. The above section on Authentication Tickets will then be obsolete.

1.2.42. Getting the Currently Logged in User

When a member of a community is logged in, you can get the `User` object instance from the running `SecurityHandler` singleton through its `CurrentUser` property.

Import Necessary Namespaces

First import the necessary namespaces that will be used to get the currently logged in user. The namespaces `StarCommunity.Core` and StarCommunity.Core.Modules.Security are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Getting the User Object

The `CurrentUser` property will return the I`User` object instance representing the user with the username contained in the authentication ticket.

```
IUser user = (IUser)StarCommunitySystem.CurrentContext.
                                    DefaultSecurity.CurrentUser;
```

1.2.43. Removing a User

Generally, removing a user in StarCommunity is a process that can be undone, optionally it can be a permanent action.

This article will show you, the developer, how to remove a user temporarily and permanently from the system.

Import Necessary Namespaces

First import the necessary namespaces that will be used to remove a user. The namespaces `StarCommunity.Core` and `using StarCommunity.Core.Modules.Security;` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Temporarily Removing the User

When temporarily removing a user we have the option to undo the action afterwards. A removed user still keeps its blog entries, forum topics, polls etc. The user will not be displayed in listings but when retrieved by id the `Removed` property of the `User` object instance will be set to `true`.

```
//Remove the user
StarCommunitySystem.CurrentContext.DefaultSecurity.RemoveUser(1234);
```

Permanently Removing the User

Permanent removal is final; all content in the StarCommunity associated with the user will be removed. The removal is made permanent by passing the `permanent` parameter as `true` to the `RemoveUser` method.

```
//Remove the user
StarCommunitySystem.CurrentContext.
                  DefaultSecurity.RemoveUser(1234, true);
```

1.2.44. Restoring a User

After a temporary removal it is possible to restore a user to an active state, this action can not be made on a permanently removed user, since the user is then no longer available in the database.

Import Necessary Namespaces

First import the necessary namespaces that will be used to restore a user. The namespaces `StarCommunity.Core` and `StarCommunity.Core.Modules.Security` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Restoring the User

To restore the user we first need to get the user's `User` object instance. We set the `Removed` property to `false` and update the user, committing our changes to the database with the `UpdateUser` method. The user is now active again and will reappear in listings and search queries.

```
//Get the user by id
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);
user = (IUser)user.Clone();

user.Removed = false;

//Update the user, restoring it to active state
StarCommunitySystem.CurrentContext.DefaultSecurity.UpdateUser(user);
```

1.2.45. Adding a User for Activation

User registration through activation by e-mail is a common way of assuring that a user's e-mail address is valid. StarCommunity solves this by temporarily storing user information in a separate part of the system, not interfering with the primary user storage. Upon activation the user data is moved to the primary user storage.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a user to the activation storage. The namespaces `StarCommunity.Core` and `StarCommunity.Core.Modules.Security` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Adding the User for Activation

In the example below we create a new `User` object instance as before, except this time we commit it with the method `AddUserToActivate`. Returned is a `Guid`, it will be used as the activation key needed to activate the user.

```
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.NewUser;
user.Alias = "Jhonny";
user.GivenName = "John";
user.SurName = "Doe";
user.BirthDate = new DateTime(1975, 3, 5);
user.EMail = "john@doe.com";
user.PassWord = "secret";
user.UserName = "john";
user.Culture =
            System.Globalization.CultureInfo.CurrentUICulture;

Guid activationGuid = StarCommunitySystem.
            CurrentContext.DefaultSecurity.AddUserToActivate(user);
```

Activating a User

After committing a user to the activation storage, we can imagine a scenario where the user recently received the activation key in an e-mail. We now have the activation `Guid`, and can activate the user.

Import Necessary Namespaces

First import the necessary namespaces that will be used to activate a user. The namespaces `StarCommunity.Core` and `StarCommunity.Core.Modules.Security` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
```

```
using StarCommunity.Core.Modules.Security;
```

Just Activating

In most cases we just want to activate the user:

```
IUser activatedUser = StarCommunitySystem.
            CurrentContext.DefaultSecurity.
            ActivateUser(
                  new Guid("3B78D829-04D5-47B0-BF5A-32C47A460FEC")
            );
```

In the above example we now got the new I`User` object instance returned with its `ID` property set. The user is now created and fully functioning.

Making Changes Before Activation

In some cases we need to make changes to the user data before activating it. We can do this by presenting the user with the option to change its information before continuing with activation.

The difference is, we retrieve the `User` object instance based on the activation key through the `GetUserToActivate` method, change the `UserName` property in this case, then commit the user to the database with the `AddUser` method. `AddUser` will recognize the user as a user from the activation storage and will remove it. The `Guid` is now no longer valid and the `ID` property is hereby the user's identifier in the primary user storage.

```
IUser activationUser = StarCommunitySystem.
            CurrentContext.DefaultSecurity.
            GetUserToActivate(
                  new Guid("3B78D829-04D5-47B0-BF5A-32C47A460FEC")
            );

activationUser.UserName = "changed";

activationUser = StarCommunitySystem.CurrentContext.
                        DefaultSecurity.AddUser(activationUser);
```

1.2.46. Adding a User to a Group

Having users as members of groups, allow you to instantly give a StarCommunity user a certain set of access rights. Access rights set on groups are automatically inherited by its members, is it users or child groups.

This article will show you, the developer, how to add a user to a group, which can be useful when registering a member of a community.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a user to a group. The namespaces `StarCommunity.Core` and `StarCommunity.Core.Modules.Security` are

described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

Attaching the Group

To attach the group to a user, simply add the `Group` object instance into the user's `GroupCollection`, visible through the `Groups` property. The group is now only attached to the user in memory, so adding or updating the user as a final step is required. In this example we commit the user by calling the `UpdateUser` method.

```
//Get the user by id
IUser user = (IUser)StarCommunitySystem.
                    CurrentContext.DefaultSecurity.GetUser(1234);
user = (IUser)user.Clone();

//Get the group by id
IGroup group = (IGroup)StarCommunitySystem.
                    CurrentContext.DefaultSecurity.GetGroup(1234);

user.Groups.Add(group);

//Update the user
StarCommunitySystem.CurrentContext.DefaultSecurity.UpdateUser(user);
```

2.2. Tags

A `Tag` can be considered to be equivalent to a word or phrase that is used by users to organize their content, commonly for the public. The use the Tag system, first import the necessary namespace:

```
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security ;
using StarCommunity.Core.Modules.Tags;
using StarCommunity.Modules.Blog; //only for this example.
```

2.2.1.Tagging an entity

The process of associating a tag with an entity item is done via the `EntityTag` class. The EntityTag enables the developer to add information of who tagged the item, available via the `Tagger` property of the EntityTag class. Keep in mind that each entity item can only be tagged with a Tag once, just as services like Flickr.

```
//Get the user by id
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

//Get the blog by id
Blog b = BlogHandler.GetBlog(1);
if (b != null)
{
    b = (Blog)b.Clone();
    b.Tags.Add(new EntityTag(new Tag("foo"), new UserAuthor(user)));
    BlogHandler.UpdateBlog(b);
}
```

2.2.2.Retrieving the tags of an entity

Retrieving the tags that an entity object has been tagged with is as simple as enumerating the `Tags` property of the `StarCommunityEntityBase`-derived object.

```
//Get the blog by id
Blog b = BlogHandler.GetBlog(1);
if (b != null)
{
foreach(EntityTag et in b.Tags)
{
        Console.Write("Tag name: "+et.Tag.Name);
        Console.Write(et.Tagger.Name);
    }
}
```

The `Tag` object is defined by its name. This makes it simple to retrieve the objects that have been tagged with "foo":

```
Tag t = new Tag("foo");
int numberOfItems = 0;
ITaggableEntity[] taggedObjects =
    t.GetItems(/*site*/null, 1, 10, out numberOfItems);
```

Items returned may be of different types; it may be Blogs, Images or Contacts that have been tagged with this the tag "foo". Common to all returned objects are that they all implement the `ITaggableEntity` interface, either directly or indirectly via the `StarCommunityEntityBase` base class.

It is also possible to retrieve the items of a specific type tagged with a tag:

```
Tag t = new Tag("foo");
int numberOfItems = 0;
ITaggableEntity[] taggedObjects =
    t.GetItems(typeof(Blog), /*site*/null, 1, 10, out numberOfItems);
```

2.2.3.Removing a tag from an entity

To remove a tag from an entity object, just call the RemoveTag method.

```
//Get the blog by id
Blog b = BlogHandler.GetBlog(1);
b.Tags.RemoveTag(”foo”);
```

2.2.4.Retrieving a tag cloud

A tag cloud is an alphabetically sorted list of the most popular tags of a certain type or globally within the system. Each tag in the list has a relative weight to the other items in the list, which is commonly used to determine the font size when rendering the tag on a web page. The StarCommunity tag system tries to retrieve a tag from each initial letter (grouping digits and non-letter characters) so that the correct number of tags is returned (defaults are configured in `Tag.config`). If the set needs to be expanded, more tags from the most popular initial letters are added to the set. If the set needs to be reduced, the least popular tags are eliminated from the set. Heuristics are applied at an early phase so that noise tags are removed (tags with very low popularity compared to the most popular tags in the set). Each item in a tag cloud encapsulates a Tag and its weight relative the other items in the tag cloud. A weight is an integer value, its lower and upper boundary configured in `Tag.config`.

```
TagCloud cloud = TagHandler.GetTagCloud();
foreach (TagCloudItem item in cloud.Items)
{
          Response.Write("<font size=\"{1}\">{0}</font> ",
                                item.Tag.Name, item.Weight);
          Response.WriteLine();
}
```

2.2.5.Implementing tag functionality on other classes

In the `StarCommunity.Modules.Blog` namespace alone there are a number of entity classes, such as `Blog`, `Entry` and `EntryComment` which all inherit `StarCommunityEntityBase`. The base class implements an ID property and the Tags property, and it is left to the entity class to implement the remaining properties.

The StarCommunity tag system allows for each entity class to have their own ID domain. However, it is required that the combination of entity type and a single integer ID uniquely specifies an instance of the entity class.

For the tag system to be able to recreate the objects there must be a EntityProvider configured for the Type that has been tagged. Providers are already configured for all relevant StarCommunity objects.

2.3. Rating

The rating system allows for rating of objects that implements the `IRatableEntity` interface. A Rating is defined as the object to rate, the rating value and the rater. Rated entities can then for example be retrieved by the average rate. The use the Rating system, first import the necessary namespace:

```
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Core.Modules.Rating;
using StarCommunity.Modules.Blog; //only for this example.
```

2.3.1. Rating an entity

In this example we use a Blog to rate. However, the similar approach is taken for all ratable entities.

```
//Get the rating user by id
IUser user = (IUser)StarCommunitySystem.
                    CurrentContext.DefaultSecurity.GetUser(1234);

//Get the blog to rate by id
Blog b = BlogHandler.GetBlog(1);

//Create a rating object with the rating value 3
IRating rating = new Rating(b, 3, new UserAuthor(user));

//Rate the blog
RatingHandler.Rate(rating);
```

2.3.2. Examine if a entity is already rated by an user

It may be of interest to see if a ratable entity has already been rated by a specific user. We call the `HasRated` method in the `RatingHandler`. In the example below, the method would return true.

```
RatingHandler.HasRated(b, new UserAuthor(user));
```

2.3.3. Retrieving ratings for an entity

There are many different overloads for getting ratings for a specific item. The example below shows how to get all ratings for Blog b, rated by a specific user

```
RatingCollection ratingCollection =
           b.GetRatings(new UserAuthor(user), 1, 10, out totalItems);
```

2.3.4.Retrieving entities based on average rating

```
//Get all entities of type Blog with an average rating of 3
int totalRatedItems = 0;
RatableEntityCollection ratedEntities =
          RatingHandler.getRatedItems(typeof(Blog), 3, 1, 10,
          out totalRatedItems);
```

2.4. Categories

The Category system allows for categorization of objects that implements the `ICategorizableEntity`interface. In this tutorial we use a `Blog` as an example.

The use the Category system, first import the necessary namespaces:

```
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Core.Modules.Categories;
using StarCommunity.Modules.Blog; //only for this example.
```

2.4.1.Add a category

Categories may be added programmatically or in the administration interface. Categories a stored in a tree structure as shown in the example below.

```
//Create a new (root) category
ICategory category = new Category("cars");

//Commit to database
ICategory rootCategory = CategoryHandler.AddCategory(category);

//Create a new sub category to root category
ICategory subCategory = new Category("volvo", rootCategory);

//Commit to database
CategoryHandler.AddCategory(subCategory);
```

2.4.2.Removea category

Categories may be removed programmatically or in the administration interface. You can get a category by id, by path or by name. in this example by path.

```
//Get the category to remove by path
ICategory category = CategoryHandler.GetCategory("cars/volvo");

//Remove the category
CategoryHandler.Removecategory(category);
```

2.4.3.Categorize an entity

A categorizable entity can have one or more categories added to it.

```
//Get the blog to categorize by id
Blog blog = (Blog)BlogHandler.GetBlog(1).Clone();

//Get the category by id
Category category = CategoryHandler.GetCategory(1);

//Add the category to the blog
blog.Categories.Add(category);

//Update the blog to commit data to the database
BlogHandler.UpdateBlog(blog);
```

2.4.4.Retrieving categories for an entity

To get a collection of all categories connected to an entity, you just call the Categories property on the categorizable entity.

```
//Get the blog to check for categories
Blog blog = BlogHandler.GetBlog(1);

//Get the categories for the blog
CategoryCollection categoryCollection = blog.Categories;
```

2.4.5.Retrieving entities based on categories

```
//Get the category for which we want entities
ICategory category = CategoryHandler.GetCategory(1);

//Add the category to the category collection
CategoryCollection categoryCollection = new CategoryCollection();
categoryCollection.Add(category);

//Get entities of type Blog that have been categorized with category
int totalItems = 0;
CategorizableEntityCollection categorizedEntities =
          CategoryHandler.GetCategorizedItems(typeof(Blog),
          categoryCollection, 1, 10, out totalItems);
```

2.5. Attributes

The Attriibute system allows for developers to setting attributes and attributes values of primitive and complex types to any object that implements the `IAttributeExtendableEntiry` interface. This provides a fast and flexible way to extend StarCommunity classes. The name of the attribute must be pre-defined in the database. This is commonly done via the administration interface. The attribute name must be unique within a type.

Note that this tutorial only describes the most elementary way to use attributes. A more object-oriented approach is to create a custom class derived from Blog that expose fixed properties. This would call for creating your own EntityProvider as explained in section 1.2.4

First import the necessary namespaces:

```
using StarCommunity.Core.Modules.Attributes;
using StarCommunity.Modules.Blog; //only for this example.
```

2.5.1.Setting attribute values

In this example we set a DateTime and a DocumentArchive attribute to a blog

```
//Get the blog for which set the attributes
Blog blog = (Blog)BlogHandler.GetBlog(1).Clone();

//Set a DateTime attribute that contains a last updated date
blog.SetAttributeValue<DateTime>("attr_last_updated", DateTime.Now);

//Get the document archive to use
DocumentArchive da = DocumentArchiveHandler.GetDocumentArchive(123);

//Set a DocumentArchive as an attribute to the blog
blog.SetAttributeValue<DocumentArchive>("attr_archive", da);
```

2.5.2.Getting attribute values

```
//Get the blog for wich we want the attriute values
Blog blog = BlogHandler.GetBlog(1);

//Get the last updated attribute
DateTime lastUpdated =
          blog.GetAttributeValue<DateTime>("attr_last_updated");

//Get the document archive attribute
DocumentArchive da =
          blog.GetAttributeValue<DocumentArchive>("attr_archive");
```

2.6. Queries

The Query system allows for dynamically creating a set of criteria that should applied to a certain type before retrieving it. A criterion can have an infinite amount of sub-criteria which in turn has their own sub-criteria. Queries like “return all Clubs with more than 10 members, with a member age range between 25 and 30 years.” are now possible, and that’s just one of the simple queries possible to compose. All relevant StarCommunity classes are retrievable and can be filtered on.

Note that this tutorial uses the Query system in the most elementary way. To read more about attributes, queries and system design, please refer to section 3.

First import the necessary namespaces:

```
Using StarSuite.Core.Modules.Sorting;
using StarCommunity.Core.Modules.Queries;
using StarCommunity.Modules.Blog; //only for this example.
```

2.6.1. Filter and and sort StarCommunity objects

In this example we want all blogs with the name “blog test” and that have 7 entries, ordered by author name ascending. Note that the criteria may be nested, as is the case with the author name.

```
//Create a new BlogQuery
BlogQuerybq = new BlogQuery()

//Initialize criterions
bq.Name = new StringCriterion();
bq.NumEntries = new IntegerCriterion();
bq.Author = new AuthorCriterion();
bq.Author.Name = new StringCriterion();

//Set values to filter on
bq.Name.Value = "Blog Test";
bq.NumEntries.Value = 7;

//Order by author name
bq.OrderBy.Add(
new CriterionSortOrder(bq.Author.Name, SortingDirection.Ascending));

//Get the filtered blog collection
BlogCollection blogs = BlogHandler.GetQueryResult(bq);
```

2.6.2.Filter on custom attributes

This is a simple example of how to filter on an attribute. In this case it's a primitive string attribute, but it could very well also be a complex attribute. If for example it was a `Forum` attribute, a `ForumCriterion` would be set instead of a `StringCriterion`. Nesting of this criterion would then also be possible.

```
//Create a blog query
BlogQuery bq = new BlogQuery();

StringCriterion strCriterion = new StringCriterion();
strCriterion.Value = "Stringvalue";
bq["stringattrib"] = strCriterion;
bq.Author = new AuthorCriterion();
bq.Author.Name = new StringCriterion();
bq.OrderBy.Add(
new CriterionSortOrder(bq.Author.Name, SortingDirection.Ascending));

BlogCollection blogs = BlogHandler.GetQueryResult(bq);
```

2.6.3.Using And / Or conditions

Criteria can be grouped and delimeted with And / Or. In this example we group the Name and NumEntries criteria, which will return Blogs with the name "Blog Test" OR with 7 entries.

```
//Create a new BlogQuery
BlogQuery bq = new BlogQuery()

//Initialize criterions
bq.Name = new StringCriterion();
bq.NumEntries = new IntegerCriterion();
bq.Author = new AuthorCriterion();
bq.Author.Name = new StringCriterion();

//Set values to filter on
bq.Name.Value = "Blog Test";
bq.NumEntries.Value = 7;

// We group Name and NumEntries and put OR inbetween
CriteriaGroup cg = new CriteriaGroup();
cg.AddCriterion(bq.Name);
cg.AddCriterion(LogicalOperator.Or, bq.NumEntries);
bq.AddCriteriaGroup(cg);

//Order by author name
bq.OrderBy.Add(
new CriterionSortOrder(bq.Author.Name, SortingDirection.Ascending));

//Get the filtered blog collection
BlogCollection blogs = BlogHandler.GetQueryResult(bq);
```

2.7. Blog

Management of blogs in StarCommunity is done through the `BlogHandler` class in the `StarCommunity.Blog` namespace. In StarCommunity, blogs are used to represent a variety of blog-like functions such as guestbooks, blogs, etc.

It is very common to have blogging functionality, guestbook functionality and similar on community sites.

This article shows you, the developer, examples of how to create this functionality with the help of the StarCommunity Framework.

Note, however, that in many of the common cases there is no need to explicitly create the `Blog`. Eg, each user's `MyPage` has both a `Blog` and a `Guestbook` property and each `Club` has a `MessageBlog` and a `NewsBlog` property. See the respective chapters for these modules for further information.

2.7.1. Adding a Blog

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
```

Adding a Blog

To add a `Blog`, we create an instance of the `Blog` class (there are several constructors available). At this point the new `Blog` exists only in memory.

```
Blog blog = new Blog("Blog name");
```

Committing the Blog Object to Database

Up until now the blog has only existed in memory, to finalize the creation of the blog we need to commit it to the system. We do this by calling the `AddBlog` method of a `BlogHandler`. Returned is the added blog, with the new unique `ID` property set. This object can now be used as a blog representation.

```
blog = BlogHandler.AddBlog(blog);
```

2.7.2. Removing a Blog

Import Necessary Namespaces

First import the necessary namespaces that will be used to remove a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
```

To remove a blog we simply need to call the `RemoveBlog` method in the `BlogHandler` with a reference to a blog as argument. This removes the entire blog permanently.

```
BlogHandler.RemoveBlog(blog);
```

2.7.3.Changing blog properties

Import Necessary Namespaces

First import the necessary namespaces that will be used to change properties of a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
```

Changing a property of a blog

Although there are constructors for the blog object that lets you set the blog presentation text right from the start, often you want to change this or other properties of a blog after the blog has been created. To do this, we fetch a blog by its unique ID, and simply change the corresponding properties on the blog object.

```
Blog blog = BlogHandler.GetBlog(17);
blog.PresentationText = "New presentation text";
```

Committing the changes to the Blog Object to Database

Up until now the changes to this blog has only existed in memory, to commit these changes to the system we need to call the `UpdateBlog` method in the `BlogHandler`.

```
BlogHandler.UpdateBlog(blog);
```

2.7.4.Adding a Blog Entry

A blog itself is only a container of sorts, the blog entries contain the actual blog content.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add an entry to a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
using StarCommunity.Core.Modules;
```

Adding a Blog Entry

To add an entry, first we need references to the blog that we want to add the entry to, and the author that has written the entry. There are several implementations of the `IAuthor` interface, such as `GuestAuthor` (the entry has no connection to any site member), `UserAuthor` (a site member is to be shown as the author) and `AnonymousAuthor` (a site member has written the entry, but has chosen to be anonymous, using a pseudonym).

As usual, there are several constructors with different sets of arguments that may be of interest.

```
IAuthor author = new GuestAuthor("Guest user");
Entry entry =
      new Entry(blog, author, "Entry title", "Entry description");
```

Committing the Blog Entry Object to Database

Up until now the new blog entry has only existed in memory, to commit the new entry to the system we need to call the `AddEntry` method in the `BlogHandler`.

Returned is the added blog entry, with the new unique `ID` property set. This object can now be used as a blog entry representation.

```
Entry = BlogHandler.AddEntry(entry);
```

2.7.5.Adding a Blog Entry with Future Publication Date

If a blogger wants to add a blog entry, but wants to have it published in specific period of time, he can do it by setting publication start and end date of the blog entry.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add an entry to a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
using StarCommunity.Core.Modules;
```

Adding a Blog Entry with Publication Dates

To add an `Entry` with specific publication dates, it is easier to use an `Entry` constructor that allows it. After crating the object, we have to commit it in the database using `BlogHandler` object.

In the example below a `GuestAuthor` adds a new blog `Entry`. This blog `Entry` will be published in 7 days, and since then will always be published (DateTime.MinValue means that this date is not considered when determining publication state of the entry).

```
Blog blog = BlogHandler.GetBlog(12);

IAuthor author = new GuestAuthor("John");
Entry entry = new Entry(blog, author, "Entry title",
  "Entry content", DateTime.Now.AddDays(7), DateTime.MinValue);
BlogHandler.AddEntry(entry);
```

Publication Dates Meanings

When creating an `Entry` (as in the example above), we need to provide both publication start and publication end date. If any of these dates will be `DateTime.MinValue`, it means that this date shall not be considered. This means that if we provide `DateTime.MinValue` as publication start date, then there is no publication start date – the entry is in published state until the publication end date. If the publication end date is `DateTime.MinValue`, it means that there is no publication end date – the entry is published since the publication start date. Analogically, if both dates are DateTime.MinValue, the entry is published since the creation date.

2.7.6.Getting Blog Entries

It is very often needed to retrieve blog entries, e.g. to list them on a web site. StarCommunity provides several overridden methods that allow entries to be retrieved for specific blog.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add an entry to a blog. The namespace `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Modules.Blog;
```

Get All Blog Entries

To retrieve all the entries of a specific blog, one just needs to call `GetEntries` method of the `Blog` class. The simplest override needs to be page number and number of items per page provided – this method is used in the example below.

```
Blog blog = BlogHandler.GetBlog(12);

// get first 100 entries from the blog
EntryCollection entries = blog.GetEntries(1, 100);
```

Getting Entries from Specific Dates and Publication Status

When listing blog entries on a webpage, it is often needed to group entries by dates. To do it, there is a `GetEntries` method overload that allows passing start and ending date of a timeframe that the entries were published (or not published).

```
Blog blog = BlogHandler.GetBlog(12);

// get fist 100 entries that are in "published" state
// between now and two weeks ahead
EntryCollection entries = blog.GetEntries(
  DateTime.Now, DateTime.Now.AddDays(14),
  EntryPublishState.Published, 1, 100);
```

2.7.7.Commenting on a Blog Entry

You may or may not want to make it possible for users to comment on posted blog entries. To do this, we start by creating a new comment.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a comment to a blog entry. The namespace `StarCommunity.Core`, `StarCommunity.Core`.Modules
This namespace contains the interface `IStarCommunityEntity` and the abstract implementation classStarCommunityEntityBase. IStarCommunityEntity implements the blueprint for tagging, attributes, rating and categorization. Also the Author classes and interfaces are located here, allowing for guests and users to identify themselves when making posts.
StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Blog`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Security;
using StarCommunity.Modules.Blog;
```

Adding a Blog Entry Comment

To add a comment to a blog entry, we need references to an entry and the author that wrote the entry. Then we simply create a new instance of the `EntryComment` class.

```
IUser user =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(17
);
IAuthor author = new UserAuthor(user);
EntryComment comment =
            new EntryComment(entry, author, "Comment title",
            "Comment description");
```

Committing the Blog Entry Comment Object to Database

Up until now the new blog entry comment has only existed in memory, to commit the new entry comment to the system we need to call the `AddEntryComment` method in the `BlogHandler`.

Returned is the added blog entry comment, with the new unique `ID` property set. This object can now be used as a blog entry comment representation.

```
comment = BlogHandler.AddEntryComment(comment);
```

2.8. Calendar

Calendar functionality in StarCommunity allows creating calendars and saving events within them, allowing community members to be up-to-date with all the community happenings.

CalendarHandler is the class that provides calendar functionality. All other classes are entity classes and are used to hold data retrieved from a database or prepared for saving to a database. Calenders are already provided for MyPage and Club classes.

2.8.1. Adding a Calendar

Before any calendar functionality can be used, a calendar has to be created – this will allow all other activity. This chapter will give you necessary knowledge to add a new calendar to the community calendars collection.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to create and add a calendar. You can go to the StarCommunity.Modules.Calendar namespace description by clicking on its name. Make sure you add the assembly as a reference, as it is mentioned in 1.1.1.

```
using StarCommunity.Modules.Calendar;
```

Create and Add a Calendar

To create a calendar, call `Calendar` class constructor providing the name for the calendar. After that, the calendar object is created and is ready to be committed in the StarCommunity database. To do it, call the `AddCalendar` method of a `CalendarHandler` object. The `AddCalendar` method returns added `Calendar` object with unique `ID` property set to a value returned from the database.

```
// Create a new calendar and get the created instance back with its
// unique id set
Calendar c = new Calendar("New Calendar");
c = CalendarHandler.AddCalendar(c);
```

2.8.2. Removing a Calendar

When a calendar is no longer needed, it can be removed from the StarCommunity database. Calendar removal is always permanent, which means that removed calendar cannot be restored as it is deleted from the database.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to remove a calendar. You can see the description of the StarCommunity.Modules.Calendar namespace by clicking on its name. Make sure you add the assembly as a reference, as it is mentioned in 1.1.1.

```
using StarCommunity.Modules.Calendar;
```

2.8.3.Remove a Calendar

To remove a calendar, a valid `Calendar` object is needed – e.g. it can be retrieved from database first. After the `Calendar` object is available, it can be removed from database by calling `RemoveCalendar` method of `CalendarHandler` object.

```
Calendar c = CalendarHandler.GetCalendar(1234);

// Remove the calendar
CalendarHandler.RemoveCalendar(c);
```

2.8.4.Adding an Event

After a calendar is created, events can be added to it. Event class describes a real event that will occur at particular date (or period), and contains information like arranger name, description, place, start date, end date etc.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add a new event. You can read descriptions of the namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Calendar` by clicking on respective name. Make sure you add the assembly as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Calendar;
```

Create a New Event

Creating a new event is a simple call to the `Event` class constructor. To call the constructor, we need to provide calendar that the event belongs to, arranger name, event author, event name, event description, event start and end date and place where it will take place. We also decide whether or not the event shall be marked as published right after adding it. After that, the event object will be created, and needs to be committed in the database using the `CalendarHandler.AddEvent` method.

```
Calendar c = CalendarHandler.GetCalendar(1234);

// Create the author of the event
IUser user = (IUser)StarCommunitySystem.CurrentContext.
                                        DefaultSecurity.CurrentUser;
UserAuthor author = new UserAuthor(user);

// Define start and end dates
```

```
DateTime startDate = new DateTime(2007, 6, 12);
DateTime endDate = new DateTime(2007, 6, 15);

// Create event object
Event ev = new Event(c, "Arranger name", author, "Event name",
"Event description", startDate, endDate, "Event place", true);

// Add event to the database
ev = CalendarHandler.AddEvent(ev);
```

The `Event` object returned from `AddEvent` method has its `ID` property set to a value returned from database.

2.8.5. Adding a Recurring Event

A recurring event is one that occurs periodically at specified dates, e.g. someone's birthdates is a recurring event, as it occurs every year at specific day of specific month. In StarCommunity, recurrence is defined using the `EventRecurrence` class. This way, if the event shall be recurrent one, all that needs to be done is to create `EventRecurrence` object and set it as a `Recurrence` property of the `Event`.

Recurrence Class Explained

The `Recurrence` class has several properties. To use the recurrence within the Calendar properly, you need to understand what each property is used for.

`Frequency` – this property defines how frequently the event is repeated. The property is of `EventRecurrenceFrequency` enumeration type:

- `DailyNumeric` – the event is repeated every `Interval` days, where `Interval` is `EventRecurrence` class property, starting from the event start date
- `DailyWeekday` – the event is repeated every weekday, starting from the start date, every `Interval` weeks (e.g. every 2nd Tuesday)
- `Weekly` – the event occurs every Nth week at days specified in the `DaysFlag` property of the `EventRecurrence` class, where N is the `Interval` specified for the `EventRecurrence` (e.g. Wednesdays and Fridays every 3 weeks)
- `MonthlyNumeric` – the event occurs every Nth month at the same day as the event start date, in intervals specified with `Interval` property of the `EventRecurrence` (e.g. Every 15th day of every 3rd month)
- `YearlyNumeric` – the event occurs every Nth year at the same day and month as the event start date, in intervals specified with `Interval` property of the `EventRecurrence` (e.g. every 31st May of every one year – someone's birthdays)

`Interval` – specifies time interval between recurrent events. `Interval` specifies only the value, unit is defined with `Frequency` property

`DaysFlag` – days at which the event occurs, used only with `Frequency` set to `EventRecurrenceFrequency.Weekly`

`StartDate` – start date of the recurrence

`EndDate` – end date of the recurrence (only `EndDate` or `MaxOccurrences` can be set for recurrence, setting one of those properties resets the second one)

`MaxOccurrences` – maximum number of recurrences that can occur before the recurrence ends (only `EndDate` or `MaxOccurrences` can be set for recurrence, setting one of those properties resets the second one)

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add a recurrent event. The namespaces `StarCommunity.Core`, `StarCommunity.Core`.Modules

This namespace contains the interface `IStarCommunityEntity` and the abstract implementation classStarCommunityEntityBase. IStarCommunityEntity implements the blueprint for tagging, attributes, rating and categorization. Also the Author classes and interfaces are located here, allowing for guests and users to identify themselves when making posts.

StarCommunity.Core.Modules.Securityand StarCommunity.Modules.Calendarare described by clicking on respective name. Make sure you add the assembly as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Calendar;
```

Create Recurrent Event

Creating recurrent event is very similar to creation of a normal event. The only difference is that a `RecurrentEvent` class object has to be created, and set to the event `Recurrence` property.

```
Calendar c = CalendarHandler.GetCalendar(1234);

// Create the author of the event
IUser user = (IUser)StarCommunitySystem.CurrentContext.
                                        DefaultSecurity.CurrentUser;
UserAuthor author = new UserAuthor(user);

// Define start and end dates - used to compute event duration
DateTime startDate = DateTime.Now;
DateTime endDate = DateTime.Now.AddDays(1);

// Create event object
Event ev = new Event(c, "Arranger name", author, "Event name",
 "Event description", startDate, endDate, "Event place", true);

// define event recurrence - every 2 weeks on Mondays and
// Wednesdays, 10 times
EventRecurrenceFrequency frequency =
            EventRecurrenceFrequency.Weekly;
int interval = 2;
DateTime recStartDate = DateTime.Now;
DateTime recEndDate = DateTime.MinValue;
int maxOccurences = 10;
EventRecurrenceDaysFlag daysFlag =
```

```
 EventRecurrenceDaysFlag.Monday | EventRecurrenceDaysFlag.Wednesday;

// Create event recurrence object
ev.Recurrence = new EventRecurrence( frequency, interval,
 recStartDate, recEndDate, maxOccurences, daysFlag, 0 );

// Add event to the database
ev = CalendarHandler.AddEvent(ev);
```

2.8.6.Inviting Users to an Event

When the event has been created, users can be invited to the event, and then e.g. invitation e-mails can be send to them. In this section we will present how to create invites and bind them to the event.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to invite users to an event. The namespaces`StarCommunity.Core`, StarCommunity.Core.Modules, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Calendar` are described by clicking on respective name. Make sure you add the assembly as a reference, mentioned in 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Calendar;
```

Invite User to an Event

To invite a user to an event, we need to have the `Event` object that the invite will be added to, and a `UserAuthor` object that defines the user we want to invite. After the `Invite` object is created, it can be committed in the database by calling `AddInvite` method of the `CalendarHandler` object.

```
Event event = CalendarHandler.GetEvent(2323);

// Get the user to invite by id
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

UserAuthor invitee = new UserAuthor(user);

// Create an invitation
Invite invite = new Invite(event, invitee);

// Add invitation to the database
invite = CalendarHandler.AddInvite(invite);
```

2.8.7.Registering upon an Event Invitation

Similar to the invitations, users can register to events by themselves – we can imagine a scenario when promotional event is prepared and only registered users will receive their individual codes to access the event (concert, show etc.).

Import Necessary Namespaces

First, import the necessary namespaces that will be used to register to an event. The namespaces`StarCommunity.Core`, StarCommunity.Core.Modules, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Calendar`are described by clicking on respective name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Calendar;
```

Registering a User to an Event

Having an event, user can register to it if:

the event `SecurityStatus` property is set to `SecurityStatus.Open` or – if the event `SecurityStatus` property is set to `SecurityStatus.Closed` – the user has been invited to the event

the day when a user wants to register is within the registration period defined for an event

the number of users that registered to the event hasn't reached the maximum number of registrations specified for the event

the user is not already registered to the event

```
Event event = CalendarHandler.GetEvent(2323);

// Get the user to register by id
IUser user = (IUser)StarCommunitySystem.
                 CurrentContext.DefaultSecurity.GetUser(1234);

UserAuthor registrant = new UserAuthor(user);

// Create a registration
Registration registration = new Registration(event, registrant);

// Register user to the event
try
{
  registration = CalendarHandler.AddRegistration(registration);
}
catch( MaxNumRegistrationsReachedException mnex )
{
  throw new Exception("Max number of registrations reached", mnex);
}
catch( AlreadyRegisteredException arex)
{
```

```
  throw new Exception("User is already registered", arex);
}
catch( RegistrationDateException rdex)
{
  throw new Exception("It is too early or too late to register",
    rdex);
}
catch( RegistrationNotInvitedException niex)
{
  throw new Exception("You have to be invited to register"
    + " to this event", niex);
}
```

2.9. Chat

The Chat Module enables users to interact with other users in real time. To join a discussion, the user simply opens a page in their browser containing Java™ applets that connect to a chat server. Discussions take place in *chat events*, and several chat events can be active simultaneously.

Chat events isolate discussions. Each chat event keeps a log of all users (and hostnames) that have entered or left the chat event, what they saidand when they said it. That is, each chat event keeps separate logs and lists of currently logged in users.

To get started, you will need some information from the chat server provider, such as the chat server's hostname. This information must be inserted in the corresponding parameters in the examples below.

2.9.1.Implementing the Chat Applets on an ASP.NET page

This section outlines the implementation of the required chat applets in an ASP.NET page. Details of how an applet is embedded in a web page differ between browsers, so we will be using a JavaScript to output the HTML that has been customized for the current browser.

```
<SCRIPT language="javascript">

function makeApplet(sArchive, sCodebase, sCode, sId, iWidth,
iHeight, aParams) {
          var _app = navigator.appName;
          if (_app == 'Microsoft Internet Explorer') {
                    document.write('<OBJECT name="'+sId+'"
id="'+sId+'"',
                                   'classid="clsid:'+
                                   '8AD9C840-044E-11D1-B3E9-
00805F499D93" ',
                                   'width="'+iWidth+'" ',
                                   'height="'+iHeight+'" ',

          'codebase="http://java.sun.com/products/'+
                                   'plugin/autodl/jinstall-1_4_2-
windows-'+
                                   'i586.cab#Version=1,4,2,0"',
          '>');
                    document.write('<PARAM NAME="code"
VALUE="'+sCode+'">');
                    document.write('<PARAM NAME="archive"
VALUE="'+
                                   sArchive+'">');
                    document.write('<PARAM NAME="codebase"
VALUE="'+
                                   sCodebase+'">');
                    document.write('<PARAM NAME="type"
VALUE="application/x-java-applet;version=1.4.2">');
                    document.write('<PARAM NAME="mayscript"
VALUE="true">');
                    document.write('<PARAM NAME="scriptable"
VALUE="true">');
```

```
                    for(var i=0; i<aParams.length; i++)
                              document.write('<PARAM
NAME="'+aParams[i][0]+
                                                  '"
VALUE="'+aParams[i][1]+'">');

                    document.write('</OBJECT>');
          }

          else /*if (_app == 'Netscape') */
          {
                    document.write('<embed ',
                              'width="'+iWidth+'" ',
                              'height="'+iHeight+'"
mayscript="true" ',
                              'type="application/x-java-
applet;version=1.4.2" ',

          'pluginspage="http://java.sun.com/j2se/1.5.0/download.htm
l" ',
                              'id="'+sId+'" name="'+sId+'"
code="'+sCode+'" codebase="'+sCodebase+'"',
                              'archive="'+sArchive+'" ');
                    for(var i=0; i<aParams.length; i++)
                              document.write(aParams[i][0]+'="'+
                                        aParams[i][1]+'" ');

                    document.write('/>');
          }
}
</SCRIPT>
```

The chat applets are split into four parts to enable the designer of the page to design a suitable framework to embed the applets into. Applets are rectangular, opaque objects that are configured as one of the following types:

Base

This (invisible) applet keeps track of the server connection(s) and does all non-GUI related tasks.

ChatWindow

This is the message window where messages from the participating users are displayed.

UserList

This applet displays a list of all the currently logged in users in the current chat event.

MessageBox

This optional applet is the input box where the user types its messages.

2.9.2.Base

The base applet is not visible to the end user, but it serves some very important functions. It is a one by one pixel applet that connects and logs in to the chat server and therefore it requires parameters that determines which server to connect to as well as the user's display name.

```
function chatNickTaken() { alert("Sorry, your alias is taken."); }

var _p = new Array();
_p[_p.length] = new Array("name", "chatBase");
_p[_p.length] = new Array("dependencies", "chatWindow chatUserList
chatMessageBox");

// IRC Server information
_p[_p.length] = new Array("ircServerAddress", "chat-server-host");
_p[_p.length] = new Array("ircServerPort", "5678");
_p[_p.length] = new Array("ircServerPassword", "pass");

// IRC User information
_p[_p.length] = new Array("ircNick", "<%=IrcNick%>");
_p[_p.length] = new Array("ircUserName", "<%=UserID%>");
_p[_p.length] = new Array("ircRealName", "<%=Name%>");
_p[_p.length] = new Array("nickTakenHandler", "chatNickTaken();");

makeApplet("IrcChat.jar", "http://chat-server-host/", "Chat.class",
"chatBase", 1, 1, _p);
```

2.9.3.ChatWindow

The chat window applet takes a number of parameters to configure its appearance and functionality. We will display a selection of them here.

```
var _p = new Array();
_p[_p.length] = new Array("name", "chatWindow");
_p[_p.length] = new Array("dependencies", "chatBase");
_p[_p.length] = new Array("ircChannels", "");
_p[_p.length] = new Array("Caption", "Main");
_p[_p.length] = new Array("eventColors", "00CC00");
_p[_p.length] = new Array("nickColors", "000000");
_p[_p.length] = new Array("backgroundColor", "ecf2ec");
_p[_p.length] = new Array("base", "chatBase");
_p[_p.length] = new Array("moduletype", "ChatWindow");
_p[_p.length] = new Array("tabpanel_borderColor","bdcbef");
_p[_p.length] = new Array("tabpanel_hideTabs", "1");
_p[_p.length] = new Array("canStartPrivateConversations", "false");
_p[_p.length] = new Array("canDisplayPersonalPage", "false");
_p[_p.length] = new Array("messagePartClassName", "Chili");
_p[_p.length] = new
Array("iconProviderURL", "/img.php/id={1}/prefix={0}");
_p[_p.length] = new Array("intromessage", "");

makeApplet("IrcChat.jar", "http://chat-server-host/",
```

```
"ChatWindow.class", "chatWindow", "100%", "100%", _params);
```

2.9.4.UserList

The user list displays the current participants in the chat event. Parameters configure for example whether the usernames are clickable or not.

```
function chatDoubleClickNick(strUserName, strNick) {
window.open("/user/"+strNick, "_blank");
}
var _p = new Array();
_p[_p.length] = new Array("name", "chatUserList");
_p[_p.length] = new Array("dependencies", "chatBase");
_p[_p.length] = new Array("ircChannels", "<%=ChannelName%>");
_p[_p.length] = new Array("Caption", "UserList");
_p[_p.length] = new Array("backgroundColor", "FFFFFF");
_p[_p.length] = new Array("base", "chatBase");
_p[_p.length] = new Array("moduletype", "UserList");
_p[_p.length] = new Array("canStartPrivateConversations", "false");

//_p[_p.length] = new Array("canDisplayPersonalPage", "false");
//_p[_p.length] = new Array("hideAnonymousUsers", "true");
_p[_p.length] = new Array("displayGroup", "false");
_p[_p.length] = new Array("displayIcons", "false");

makeApplet("IrcChat.jar", "http://chat-server-host/",
"ChatUserList.class", "chatUserList", "100%", "100%", _p);
```

2.9.5.MessageBox

This applet displays a text box where the user inputs its next message. It can be used in one of three modes: displayed, hidden or absent. In the hidden mode, put an alternate input method, such as a normal `input type="text"` in your document and call the `sendMessage(message)` method with JavaScript.

When the MessageBox applet is absent, it is because the current user should only be able to monitor the chat event but not be able to participate in it. If the MessageBox applet is absent, do not forget to adjust the Base applet's dependencies accordingly.

```
// Used if you need to send a message to the channel
// with JavaScript.
function sendMessage(m) {
          var app = document.getElementById("chatMessageBox");
          if (app) {
                    if (m=="") app.sendMessage();
                    else app.sendMessage(m);
          }
}

var _p = new Array();
_p[_p.length] = new Array("name", "chatMessageBox");
_p[_p.length] = new Array("dependencies","chatBase");
_p[_p.length] = new Array("ircChannels","<%=ChannelName%>");
<% if ((!IsModerator) && (IsModerated) && (!IsVipUser)) { %>
_p[_p.length] = new Array("sendToIrcChannel",
```

```
"<%=ModeratorChannelName%>");
<%
}
%>
_p[_p.length] = new Array("Caption",  "MessageBox");
_p[_p.length] = new Array("backgroundColor", "FFFFFF");
_p[_p.length] = new Array("base", "chatBase");
_p[_p.length] = new Array("moduletype", "MessageBox");
_p[_p.length] = new Array("maxMessageLength", "400");

makeApplet("IrcChat.jar", "http://chat-server-host/",
"ChatMessageBox.class", "chatMessageBox", 1, 1, _p);
```

2.10. Club

Clubs are mini-communities within a community. A club has a separate member list, club news, forums and image galleries etc. Clubs can be treated in different ways; they can be hidden, which means that only people who know about them can become members. It can have different security states: closed or open. Closed clubs require approval from its owner before allowing more members, while open clubs are free to join by anyone. Clubs can also be created and wait for subsequent approval from an administrator, or the community can allow for free creation of clubs.

2.10.1. Adding a Club

Adding a club through the API is useful when users want to create their own clubs based on topics.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a club. The namespaces `StarCommunity.Modules.Club`, StarCommunity.Cre.Modules.Security and `StarCommunity.Core` are described by clicking on their respective names. Make sure you add the assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Club;
```

Creating the Club

In this example we start by getting the `User` object instances of the users we want as the creator and owner of the club. Secondly we have decided to set this club as approved, visible and with security status set to `SecurityStatus.Closed`.

```
//Get the creator by id
IUser creator = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

//Get the owner by id
IUser owner = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(12345);

bool isApproved = false;
bool isHidden = false;

Club club = new Club("Club name",
                       "Club description", isApproved,
                       null, null, SecurityStatus.Closed,
                       "Reason for creation", creator, owner,
                       null, isHidden);


club = ClubHandler.AddClub(club);
```

Finally we add the club to database by calling the `AddClub` method. The new `Club` object instance with its `ID` property set is returned.

2.10.2. Removing a Club

Removing a club can be done either temporarily or permanently. As with users, a permanent removal means there is no way of undoing the action, while a temporary removal only results in the club not appearing in listings and search queries.

Importing Necessary Namespaces

First import the necessary namespaces that will be used to remove a club. The namespace `StarCommunity.Modules.Club` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Club;
```

Removing a Club

To temporarily remove a club get the `Club` object instance id, and send it as a parameter to the `RemoveClub` method. If we would now try to get the `Club` object instance with the `GetClub` method, the club's `Removed` property would be set to `true`.

```
//Get the club by id
Club club = ClubHandler.GetClub(1234);

//Temporarily remove the club
ClubHandler.RemoveClub(club);
```

Permanently Removing a Club

To permanently remove a club we start off in the same way as we did with temporary removal. The difference is when we call the `RemoveClub` method, since this time we pass the `permanent` parameter as `true`. If you try to retrieve the `Club` object instance now, using the `GetClub` method, it will return `null` since the club no longer exists in the database.

```
//Get the club by id
Club club = ClubHandler.GetClub(1234);

//Permanently remove the club
ClubHandler.RemoveClub(club, true);
```

2.10.3. Adding Club Members

Becoming a member of a club is necessary if you want to share its information. A club owner does not have to go through the process of becoming a member after creating a club, since owners are automatically added to the member's list.

This article will show you, the developer, how to add a user as a member of a club and what the different results will be if the club is closed or open.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a club member. The namespaces `StarCommunity.Modules.Club`, StarCommunity.Core.Modules.Security and `StarCommunity.Core` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Club;
```

Adding a Member

To add a member we first get the club we want to become member of by its id, and then the user that is to become a member by its id.

We create a `Membership` object instance and supply the club and user as arguments to its constructor. When we create the `Membership` object instance, the membership will be set to `MembershipType.Applied` if the club is set to `SecurityStatus.Closed`. This means that if we use this default behavior, the membership will have to be applied by an administrator or the club owner before it is valid. Finally we call the `AddMembership` method to store the membership in the database.

```
//Get the club by id
Club club = ClubHandler.GetClub(1234);

//Get the new member by id
IUser user = (IUser)StarCommunitySystem.
                 CurrentContext.DefaultSecurity.GetUser(1234);


Membership m = new Membership(user, club,
                              "Reason why I want to join");

// Add the membership to database
m = ClubHandler.AddMembership(m);
```

2.10.4. Adding Club Ads

Club ads are used to promote a club in a community, either with images or text. In this article we explain how to create a new Club Ad with an ImageGallery Image attached to it.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a Club Ad. The namespaces `StarCommunity.Modules.Club`, StarCommunity.Core.Modules.Security and `StarCommunity.Core` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Mmodules.Security;
using StarCommunity.Modules.Club;
```

Adding an Ad with an Image

To create a Club Ad from an image located at e.g. `C:\Image.jpg` requires the use of the ImageGallery Module. Now we get the club we are making an ad for by its id, then we create an `Ad` object instance. We have decided to set the ad as approved from the start; this means that it does not need approval from an administrator. Finally we add the `Ad` object instance to the database by calling the `AddAd` method.

```
System.IO.FileStream fs =
            new System.IO.FileStream(@"C:\Image.jpg",
            System.IO.FileMode.Open);

using(fs)
{
      //Get the image uploader by id
      IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

           StarCommunity.Modules.ImageGallery.Image image =
                     new StarCommunity.Modules.
                  ImageGallery.Image("Name", "Description", fs,
                  uploader);

      //Get the club by id
      Club club = ClubHandler.GetClub(1234);
      bool isApproved = true;

           Ad ad = new Ad(club, "Test header",
                        "Test body", isApproved, image);

           ad = ClubHandler.AddAd(ad);
}
```

2.10.5. Setting Club Keywords

Keywords for clubs are used to give better results out of search queries. This article explains how you bind a keyword to a club.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a club keyword. The namespace `StarCommunity.Modules.Club` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Club;
```

Adding a Keyword

As usual we start by creating a new `ClubHandler` object instance, we then get the club we want to set the keyword on, by its id. Finally we create a new `Keyword` object instance and supply the club together with the keyword as parameters to the constructor. Finally we add the keyword to the database by calling the `AddKeyword` method. Search queries on this keyword will not return this club.

```
//Get the club by id
Club club = ClubHandler.GetClub(1234);

Keyword keyword = new Keyword(club, "Keyword Text");
keyword = ClubHandler.AddKeyword(keyword);
```

2.11. ConnectionLink

ConnectionLink uses data from `StarCommunity.Modules.Contact` and performs Breadth First Search (BFS) algorithms to decide the shortest path between 2 users.

2.11.1. Getting the Shortest Path

Use the `ConnectionLinkHandler` to get aUserCollection containing `userA` and the users representing the shortest path to `userB` based on contact relations.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage the connection link and users. The namespace`StarCommunity.Modules.ConnectionLink`, `StarCommunity.Core` andStarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.ConnectionLink;
```

Get the two users, `userA` and `userB`, to compare.

```
//Get the userA and userB by id
IUser userA =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

IUser userB =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
35);
```

Get the `UserCollection` with the users representing the shortest path from `userA` to `userB`

```
UserCollection connections =
            ConnectionLinklHandler.GetShortestPath( userA, userB );
```

2.12. Contact

Management of contacts is done through the `ContactHandler` class in the `StarCommunity.Contact` namespace. The connection between the user and its contact relations is done via the `ContactContainer` class. However, a ContactContainer is created automatically for a user upon user creation, for each site in the system, and it is normally not done by the developer.

2.12.1. Adding a Contact Relation

To add a `ContactRelation`, first create an instance of the `ContactRelation` class (there are several constructors available). The `ContactRelation` constructor needs the `ContactContainer` for the user that is adding the contact relation (`userA`), the user to be added to the contact list (`userB`) and a `ContactType`. The contact type can be either `ContactType.Request` or `ContactType.Contact.ContactType.Request` is used when the contact relation must be approved by `userB` and `ContactType.Contact` is used when the contact relation should come into effect immediately.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contacts and users. The namespace`StarCommunity.Modules.Contact`, `StarCommunity.Modules.MyPage`,`StarCommunity.Core` and StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Contact;
using StarCommunity.modules.MyPage;
```

First we must get the `ContactContainer` for `userA`. This can be done by using the `ContactHandler`, but in most cases it is accessed by the `Contact` propertyin the `MyPage` class.

```
// Get the userA by id
IUser userA =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

// Get the contact container for userA via my page
MyPage myPageA = MyPageHandler.GetMyPage(userA);
ContactContainer contactContainerA = myPageA.Contact;
```

The `contactContainerA` belonging to `userA` can now be used for creating a `ContactRelation to userB`.

```
//Get the userB by id
IUser userB =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
35);

ContactRelation contactRelation =
new ContactRelation(contactContainerA, userB, ContactType.Contact);
```

At this point the new `ContactRelation` object exists only in memory, to commit it to the database and get the unique `ID` property set, we need to add it using the `ContactHandler`.

```
//Commit the contact relation to database
contactRelation =
      ContactHandler.AddContactRelation(contactRelation);
```

Note that the `AddContactRelation` method always returns the committed object with the unique `ID` property set. Depending on the Configuration File, a corresponding contact relation of `ContactType.Contact` from `userB` to `userA` may automatically have been created. This is the most common way to create contact relations where no contact approval is needed.

2.12.2. Removing a Contact Relation

First we need to get the `ContactRelation` object to be removed. This is done via the `ContactContainer` for the removing user (`userA`). The `ContactContainer` can be accessed via the `ContactHandler` but is most often accessed by the `Contact` property in the `MyPage` class.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contacts and users. The namespaces`StarCommunity.Modules.Contact, StarCommunity.Core and` StarCommunity.Core.Modules.Security are described by clicking on their names. Make sure you add the assemblies as a reference, as mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Contact;
using StarCommunity.Modules.MyPage;
```

Get the contact relation between `userA` and `userB`.

```
// Get the userA by id
IUser userA =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

MyPage myPageA = MyPageHandler.GetMyPage(userA);
ContactContainer contactContainerA = myPageA.Contact;
```

```
// Get the userB by id
IUser userB =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
35);

// Get the contact relation for userA to userB
ContactRelation contactRelation =
                  contactContainerA.GetContactRelation(userB);
```

Now we have the `ContactRelation` to be removed and we use the `ContactHandler` to remove it.

```
ContactHandler.RemoveContactRelation(contactRelation);
```

Depending on Configuration File the corresponding `ContactRelation` from `userB` to `userA` may automatically have been removed if the `ContactType` is set to `ContactType.Contact`.

2.12.3. Approving a Contact Relation

Approving a ContactRelation code is basically about setting the `ContactType.Request` to `ContactType.Contact`. The following example shows how `userA` is requesting a contact relation to `userB`, which `userB` approves, resulting in that a `ContactRelation` of `ContactType.Contact` is created from `userA` to `userB` and consequently from `userB` to `userA`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contacts and users. The namespace`StarCommunity.Modules.Contact, StarCommunity.Core and` StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Contact;
using StarCommunity.Modules.MyPage;
```

First create a `ContactRelation` from `userA` to `userB` of `ContactType.Request`. This is done by getting the ContactContainercode for `userA`, usually accessed from the

MyPage class.

```
// Get userA by id
IUser userA =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

// Get userA contact container via my page
MyPage myPageA = MyPageHandler.GetMyPage(userA);
ContactContainer contactContainer = myPageA.Contact;
```

Then create a ContactRelation from userA to userB of ContactType.Request and use the ContactHandler to add it and thereby commiting it to the database.

```
// Get userB by id
IUser userB =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
35);

// Create a contact relation between userA and userB of type Request
ContactRelation contactRelationA =
new ContactRelation(contactContainer, userB, ContactType.Request);

// Commit the contact relation to database
contactRelation =
            ContactHandler.AddContactRelation(contactRelation);
```

userA now has a one-way relation of type ContactType.Request to userB. To create a two-way relation of ContactType.Contact, userB needs to approve the request. This is done by updating the ContactRelationcode to be of ContactType.Contact. First we need to get the ContactRelation that should be updateed. This is done via the ContactHandler class:

```
ContactRelation contactRelation =
                 ContactHandler.GetContactRelation(userA, userB);
```

Then we change the ContactType property from ContactType.Request to ContactType.Contact.

```
contactRelation.ContactType = ContactType.Contact;
```

The changes to the object are now only represented in memory. We need to update the object via the ContactHandler class to commit the object state to the database.

```
ContactHandler.UpdateContactRelation(contactRelation);
```

Optionally, but commonly, you can now add a relation from `userB` to `userA` to get a two-way relation between `userA` and `userB`. First we get the `ContactContainer` for `userB` via the `userB` MyPage:

```
MyPage myPageB = MyPageHandler.GetMyPage(userB);
ContactContainer contactContainerB = myPageB.Contact;
```

Then create a new `ContactRelation` to `userA` and add it to commit the object to the database.

```
ContactRelation contactRelation =
new ContactRelation(contactContainerB, userA, ContactType.Contact);
contactRelation =
            ContactHandler.AddContactRelation(contactRelation);
```

The `AddContactRelation` method returns the committed object with the unique `ID` property set.

2.12.4. ContactRelationCollections and Perspectives.

To get a `ContactRelationCollection` we call the `GetContactRelations` method in the `ContactContainer` supplying the `ContactType` we are interested in. Since a contact relation request can be either directed towards the current user or a request from the current user towards another user, we need to introduce the enum `Perspective`. `Perspective` values can be either `Perspective.ToMe` or `Perspective.FromMe`. The following samples show the use of perspectives.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contacts and users. The namespace`StarCommunity.Modules.Contact, StarCommunity.Core and` StarCommunity.Core.Modules.Securityis described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Contact;
using StarCommunity.Modules.MyPage;
```

First we get the `userAContactContainer` via the `MyPage` class.

```
// Get userA by id
IUser userA =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
```

```
34);

// Get userA contact container via my page
MyPage myPageA = MyPageHandler.GetMyPage(userA);
ContactContainer contactContainerA = myPageA.Contact;
```

Get all contact relations of `ContactType.Contact` belonging to `userA`'s `ContactContainer`.

```
int totalHits = 0 ;
ContactRelationCollection crCollection =
      contactContainerA.GetContacts(ContactType.Contact,
      Perspective.FromMe, 1, 20, out totalHits,
      new
      ContactRelationSortOrder(ContactRelationSortField.ContactAlias
      , SortDirection.Ascending));
```

We are getting page 1 with 20 items per page and sorting on alias ascending.

Get all pending contact relation requests from other users to `userA`. Note that we use `Perspective.ToMe`.

```
int totalHits = 0 ;
ContactRelationCollection crCollection =
      contactContainerA.GetContacts(ContactType.Request,
      Perspective.ToMe, 1, 20, out totalHits,
      new ContactRelationSortOrder
      (ContactRelationSortField.ContactAlias,
      SortDirection.Ascending));
```

Get all pending requests made by `userA` to other users. The only thing we need to change is `Perspective.FromMe`.

```
int totalHits = 0 ;
ContactRelationCollection crCollection =
      contactContainerA.GetContacts(ContactType.Request,
      Perspective.FromMe, 1, 20, out totalHits,
      new ContactRelationSortOrder
      (ContactRelationSortField.ContactAlias,
      SortDirection.Ascending));
```

2.12.5. Configuration File

ELEMENT NAME	TYPE	DESCRIPTION
ReverseAdd	Boolean	If set to true, when a ContactRelation of type Contact from userA to userB is added, a corresponding ContactRelation from userB to userA will be added automatically. Useful when your community doesn't use approval steps for adding contact relations.
ReverseRemove	Boolean	If set to true, when a ContactRelation of type Contact from userA to userB is removed, the corresponding ContactRelation from userB to userA will be removed automatically.

2.13. Contest

Contest management is done through the `ContestHandler` class in the`StarCommunity.Modules.Contest`namespace. Contests are typically created and managed by an administrator in the administration interface and not programmatically by a developer. In this tutorial only actions you typically need to do front-end are shown. See the *StarCommunity User Manual* for further information on how to create and manage contests.

2.13.1. Get Contests

We use the `ContestHandler` class to return a `ContestCollection` with all existing contests. In this case we get page 1 with 20 items per page, sorted on creation date ascending.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contests. The namespace `StarCommunity.Modules.Contest` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Contest;
```

Get the contest collection via the `ContestHandler`:

```
int totalHits = 0 ;
ContestCollection cCollection =
            ContestHandler.GetContests(1, 20, out totalHits,
            new ContestSortOrder(ContestSortField.Created,
            SortDirection.Ascending));
```

2.13.2. Get Contest Questions

Contest questions can be of 2 different types that all inherit the `Question` base class. The types are `AlternativeQuestion`and`TextQuestion`. The `AlternativeQuestion` is in turned to `SingleAlternativeQuestion` and `MultipleAlternativeQuestion` classes.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contests. The namespace `StarCommunity.Modules.Contest` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Contest;
```

To the contest questions we first need the contest. This is done by calling the `GetContest` method in the `ContestHandler` class.

```
Contest contest = ContestHandler.GetContest( 1234 );
```

Then we get the contest questions collection via the `Questions` property in the `Contest` class.

```
QuestionCollection qCollection = contest.Questions;
```

If the Questioncode is of type `AlternativeQuestion` you typically want to get the alternatives. This is done via the `Alternatives` property of the `AlternativeQuestion` class. Note that you need to examine the Questioncode object in the `QuestionCollection` to decide whether it is an `AlternativeQuestion` before accessing the `Alternatives` property. To get the `Question` object we use the `ContestHandler` class.

```
Question question = cHandler.GetQuestion( 1234 );
if( question is AlternativeQuestion )
{
      AlternativeCollection altCollection =
            ((AlternativeQuestion)question).Alternatives;
}
```

Note: The `question` object is often accessible during `ItemDataBound` for repeaters and data lists in a user control where a `QuestionCollection` typically acts as the datasource. The `GetQuestion` method is therefore seldom used in this context.

2.13.3. Add Contest Submission

In the following sample we assume that we have a question of type `SingleAlternativeQuestion`. To submit a contest submission, we first need to populate an `AnswerCollection`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contests. The namespace `StarCommunity.Modules.Contest` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Contest;
```

First we need to get the contests for which we want to submit our answers.

```
Contest contest = ContestHandler.GetContest( 1234 );
```

We initialize the answer collection.

```
AnswerCollection answers = new AnswerCollection();
```

We get an alternative selected by the user via the `ContestHandler` class.

```
Alternative alternative = cHandler.GetAlternative( 1234 );
```

Then we get `SingleAlternativeAnswer` from the selected alternative and add it to `AnswerCollectionAnswer.`

```
SingleAlternativeAnswer saa =
                        SingleAlternativeAnswer( alternative );
answers.Add( saa );
```

Create a contest submission for the user.

```
Submissin submission =
            new Submission(contest, answers, user, "John", "Doe",
            "Address", "Zip", "City", "Email");
```

At this point the submission only exists in memory. To commit it to the database we use the `AddSubmission` method in the `ContestHandler` class.

```
ContestHandler.AddSubmission( submission );
```

2.13.4. Get winners

Winners are typically selected by an administrator in the administration interface. However, you often want to get the winners for a contest for displaying them on your community.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage contests. The namespace`StarCommunity.Modules.Contest` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Contest;
```

First we need to get the `Contest`. This is done via the `ContestHandler` class.

```
Contest contest = ContestHandler.GetContest( 1234 );
```

To get a `UserCollection` of winners for the contest, we access the `Winners` property in the `Contest` class.

```
UserCollection winners = contest.Winners;
```

2.14. DirectMessage

Message management is done through the `DirectMessageHandler` class in the `StarCommunity.Modules.DirectMessage`namespace. Three root folders for each site are created automatically for each user upon user creation: Inbox, Sent and Draft. The root folders are accessible via the `DirectMessageContainer` class, which in turn is often accessed via the `MyPage` class.

2.14.1. Send a Message

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage messages and users. The namespace `StarCommunity.Modules.DirectMessage, StarCommunity.Core and` StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.DirectMessage;
using StarCommunity.Modules.MyPage;
```

We start by creating a new `Message` providing the user who is sending the message.

```
//Get the sender user by id
IUser senderUser = (IUser)StarCommunitySystem.CurrentContext.
                                        DefaultSecurity.GetUser(1234);

//Create a new message
Message message =
          new Message("message subject", "message body",
          senderUser, null);
```

Create recipients and add them to the message recipients list via the `Recipients` property of the `Message`. To create a `MessageReceiver` we need a user and can optionally specify a folder where we should put the message. In this case we put it in the recipient system folder inbox. This folder is default if no folder is supplied. In this example we get the inbox via the recipients MyPage

```
//Get the recipient user by id
IUser recipientUser = (IUser)StarCommunitySystem.CurrentContext.
                                        DefaultSecurity.GetUser(1235);

MyPage recipientMyPage = MyPageHandler.GetMyPage(recipientUser);
message.Recipients.Add(new MessageRecipient( MyPageUser,
recipientMyPage.DirectMessage.
GetSystemFolder(SystemFolderType.Inbox)));
```

Now we are ready to send the message. This is done using the `DirectMessageHandler`.

```
DirectMeessageHandler.SendMessage( message );
```

Now the message is in the receiver's Inbox folder. If you pass the second parameter "copyToFolder" you can have a copy of the message delivered to for example the sender's Sent-folder.

2.14.2. Removing Messages

If you have opened for the possibility to have more than one `MessageRecipient` in your community, you cannot simply remove the entire message since all receivers "share" the same message. Instead you remove the message from the folderin question. When the message is removed from all folders, the actual `Message` is removed automatically. In the following sample we send a message from `senderUser` to two receivers (`recipient1` and `recipient2`) then remove the message from their Inboxes.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage messages and users. The namespace `StarCommunity.Modules.DirectMessage, StarCommunity.Core and` StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.DirectMessage;
using StarCommunity.Modules.MyPage;
```

First we create the message, add receivers and send it:

```
//Get the sender user by id
IUser senderUser = (IUser)StarCommunitySystem.CurrentContext.
                                    DefaultSecurity.GetUser(1234);

//Get 2 recipients users by id
IUser recipientUser1 =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);
IUser recipientUser2 =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
35);

//Create and send the message to the 2 recipients
Message message = new Message("Test subject", "test body",
                                                    senderUser);
MessageRecipient recipient1 = new MessageRecipient
                                        (recipientUser1);
MessageReceiver recipient2 = new MessageRecipient
                                        (recipientUser2);
```

```
message.Recipients.Add(recipient1);
message.Recipients.Add(recipient2);

DirectMessageHandler.SendMessage( message );
```

`recipient1` now removes the message from his/hers Inbox folder. First we need to get the folder. We get this via the recipient MyPage.

```
MyPage recipient1MyPage = MyPageHandler.GetMyPage(recipientUser1);
SystemFolder inbox1 =
          recipientMyPage1.DirectMessage.
          GetSystemFolder(SystemFolderType.Inbox)
```

We call the `RemoveMessage` method in the `DirectMessageHandler` class.

```
DirectMessageHandler.RemoveMessage(message, inbox1);
```

Note that the actual message still exists since `receiver2` still have it in its Inbox. If `receiver2` also removes the message and `senderUser` removes it from his/hercopy folder, the message itself will be automatically removed.

If you want to remove the message from all folders, you can use the `RemoveMessage`overload that only takes a `Message` as an argument.

```
DirectMessageHandler.RemoveMessage( message );
```

2.14.3. Listing Messages in Folders

All messages are located in one of the user's three system folders: Inbox, Sent and Draft (or their subfolders). The `MessageCollections` are therefore accessible via the `Folder` class. The following sample shows how to retrieve a `MessageCollection` from the root folder Inbox.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage messages and users. The namespace `StarCommunity.Modules.DirectMessage`, `StarCommunity.Core` and StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.DirectMessage;
using StarCommunity.Modules.MyPage;
```

First we need the `DirectMessageContainer` for the user who's Inbox we want to list. We access it via the users `MyPage`.

```
//Get the user by id
IUser user =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

//Get the direct message container via user my page
MyPage myPage = MyPageHandler.GetMyPage( user );
DirectMessageContainer dmc = myPage.DirectMessage;
```

The root folders are accessible via the `DirectMessageContainer`. In this case we want the Inbox folder:

```
Folder inbox = dmc.GetSystemFolder(SystemFolderType.Inbox)
```

Now we call the `GetMessages` method to get the `MessageCollection`. In this case we get page 1 with 20 items per page, sorted by creation date ascending.

```
MessageCollection messages = inbox.GetMessages(1, 20,
            new DirectMessageSortOrder
            (DirectMessageSortField.DateCreated,
            SortDirection.Ascending));
```

2.14.4. Flag a Message as read

When a receiver is reading the message you often want to visualize that the message has been read. This is done by the updating the `MessageRecipientHasRead` To get the message recipient we need to specify a message and a user.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage messages and users. The namespace `StarCommunity.Modules.DirectMessage`, `StarCommunity.Core` and StarCommunity.Core.Modules.Security is described by clicking on their names. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.DirectMessage;
```

```
//Get the message
Message message = (Message)DirectMessageHandler.GetMessage(1234);
```

```
//Get the recipient user by id
IUser recipientUser =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

//Get the message recipient
MessageRecipient recipient =
           (MessageRecipient)DirectMessageHandler.GetRecipient(messa
ge, recipientUser).Clone();

//Update recipient and commit to database
recipient.HasRead = true;
DirectMessageHandler.UpdateRecipient(recipient);
```

The message is now flagged as read by the `recipientUser`and the read date has been automatically set and can be accessed via the `ReadDate` property in the `MessageRecipient` class

2.15. Document Archive

The document archive is used for file sharing purposes between community members. A document archive is automatically created on `MyPage` and Club creation and accessible via the `DocumentArchive` property. Document archives can also be created as stand-alone archives.

2.15.1. Add a Document Archive

Document archives automatically exist for `MyPage` and `Club`. However, if you need a stand-alone archive you simply add a new document archive, this is done via the `DocumentArchiveHandler`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage document archives. The namespace`StarCommunity.Modules.DocumentArchive`is described by clicking on the name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.DocumentArchive;
```

Create a new`DocumentArchive:`

```
DocumentArchive da = new DocumentArchive("Name", "Description");
```

At this point, the `DocumentArchive da` only exists in memory. To commit it to database you call the `AddDocumentArchive` method in the `DocumentArchiveHandler` class.

```
da = DocumentArchiveHandler.AddDocumentArchive(da);
```

Note that the `AddDocumentArchive` method returns the committed object with the unique `ID` property set.

2.15.2. Remove a Document Archive

Stand-alone archives and all of their content can be removed. This is done via the `DocumentArchiveHandler RemoveDocumentArchive` method. First we need the document archive to be removed.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage the document archive. The namespace `StarCommunity.Modules.DocumentArchive` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.DocumentArchive;
```

We use the `GetDocumentArchive` method in the `DocumentArchiveHandler` to get the archive to be removed.

```
da = DocumentArchiveHandler.GetDocumentArchive( 1234 );
```

Then we remove it:

```
DocumentArchiveHandler.RemoveDocumentArchive( da );
```

2.15.3. Add a Document

To add a document to a `DocumentArchive` we need a document. The document is constructed with a `DocumentArchive`, a `User` and a file `Stream`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage document archives and users. The namespace`StarCommunity.Modules.DocumentArchive`, `StarCommunity.Core and` StarCommunity.Core.Modules.Security is described by clicking on their names. We also need to import `System.IO` to use the `Stream` class. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using System.IO;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.DocumentArchive;
using StarCommunity.Modules.MyPage;
```

First we get the document archive in which we want to add our document. In this case it is the document archive that has been automatically created for us upon MyPage creation.

```
//Get the user by id
IUser user =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

//Get the document archive for user via my page
MyPage myPage = MyPageHandler.GetMyPage( user );
DocumentArchive da = myPage.DocumentArchive;
```

Now we can create the document by providing the user that is uploading the file the document archive and a file stream. In this case the archive owner is the same user as the uploader.

```
//Get the file stream
FileStream stream = new FileStream("foo.doc", FileMode.Open);
```

```
//Create the document
Document doc =
      new Document( "foo.doc", "foo description", da, user, stream);
```

Now we add the `Document doc` using the `DocumentArchiveHandler` and at the same time committ it to database, until now it has only been represented in memory.

```
doc = DocumentArchiveHandler.AddDocument( doc );
```

`doc` now contains the committed object with the unique `ID` property set.

2.15.4. Update a Document

To update an existing document we first need the document. The document can be retrieved via the `DocumentArchiveHandler`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage the document archive. The namespace `StarCommunity.Modules.DocumentArchive` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.DocumentArchive;
```

Retrieve the document archive using the `GetDocument` method in the `DocumentArchiveHandler` class.

```
doc = (Document)DocumentArchiveHandler.GetDocument( 1234 ).Clone();
```

Now we update the `doc` object, changing its description:

```
doc.Description = "An updated description";
```

Note that the object is now only modified in memory. To commit the changes to database, we use the `UpdateDocument` method in the `DocumentArchiveHandler` class.

```
DocumentArchiveHandler.UpdateDocument( doc );
```

2.15.5. Remove a Document

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage the document archive. The namespace `StarCommunity.Modules.DocumentArchive` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.DocumentArchive;
```

Before we can remove a document from a document archive, we need to get the document to remove. This is done via the `DocumentArchiveHandler`.

```
doc = DocumentArchiveHandler.GetDocument( 1234 );
```

We can then remove it.

```
DocumentArchiveHandler.RemoveDocument( doc );
```

2.15.6. Configuration File

ELEMENT NAME	TYPE	DESCRIPTION
PhysicalPath	String	The physical path to where document archive files should be stored.
VirtualPath	String	The virtual path to where document archive files should be stored.

2.16. Expert

StarCommunity Expert module provides functionality enabling users to ask questions that can be answered by the domain experts. Experts do not have to be community members to provide answers to the questions.

2.16.1. Add an Expert

To fully use the Expert functionality, at least one Expert has to be added to the community to answer the questions. In this section, we will focus on adding an Expert that is not a community member – its only job in the community is to answer questions.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add an expert. The namespaces `StarSuite.Core`, and `StarCommunity.Modules.Expert` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in 1.1.1.

```
using StarSuite.Core;
using StarCommunity.Modules.Expert;
```

Create a New Expert

Creating a new expert is a simple call to the `Expert` class constructor. The `Expert` class constructor allows for providing several properties describing an expert: first and last name, e-mail address, general description, qualifications, home page, phone, status and assigned site within the community.

When the `Expert` object has been created, it has to be committed to database using the `AddExpert` method of the `ExpertHandler` class object.

```
// Get site that the expert will be assigned to
StarSuite.Core.Modules.ISite site =
StarSuite.Core.SiteHandler.GetSite(1);

// create a new Expert
ExpertBase expert = new Expert("John", "Doe", "john@doe.com",
"Description", "Qualifications", "555-55-55",
  "http://johndoe.expert.com", ExpertStatus.Active, site);

// save expert in the database
expert = ExpertHandler.AddExpert(expert);
```

2.16.2. Add a Member Expert

An Expert does not have to be completely independent – a community member can be an expert too. The only difference is that when creating a member expert, there is no need to provide first name, last name and e-mail of an expert as these values are already known.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add an expert. The namespaces`StarSuite.Core`, `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Expert` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Core.Modules;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Expert;
```

Create a New Member Expert

Creating a new member expert is a simple call to the `ExpertMember` class constructor. The `ExpertMember` class constructor allows for providing several properties describing a member expert: the user that is to become an expert, general description, qualifications, home page, phone, status and assigned site.

When the `ExpertMember` object has been created, it has to be committed in the database using the `AddExpert` method of the `ExpertHandler` class object.

```
// Get site that the expert will be assigned to
StarSuite.Core.Modules.ISite site =
StarSuite.Core.SiteHandler.GetSite(1);

// Get current user to become an expert
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// create a new Expert
ExpertBase expert = new ExpertMember(user,
"Description", "Qualifications", "555-55-55",
  "http://johndoe.expert.com", ExpertStatus.Active, site);

// save expert in the database
expert = ExpertHandler.AddExpert(expert);
```

With `ExpertMember` it has to be remembered that `GivenName`, `SurName` and `EMail` properties of the class are read only. Each of these properties has a setter, but it throws `NotSupportedException`, as the values are taken from the injected User instance. To change them, change the `ExpertMember.User` properties.

2.16.3. Remove an Expert

When the existing `Expert` or `ExpertMember` does not want to or cannot be an expert any more, he can be removed from the StarCommunity. This section briefly explains how to do that.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to remove an expert. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Remove an Expert

Removing an expert is a very simple operation – all that is needed is to have an `Expert` or `ExpertMember` object, and call the `RemoveExpert` method of the `ExpertHandler` class. Removing an expert means that he will no longer be able to login, view or answer questions etc. If the expert is `ExpertMember`, the underlying user is not deleted – he is simply no longer an expert within the community, but is still a valid user of the community.

```
// Get an expert to remove
ExpertBase expert = ExpertHandler.GetExpert(234);

// remove the expert
ExpertHandler.RemoveExpert(expert);
```

2.16.4. See if a User is an Expert

It is often needed in the application to display different user interface depending on whether the currently logged in user has some capabilities or not. The same applies to the Expert functionality – when a user is an expert, there can be a need to display questions assigned to him, or enable user interface features to answer questions. This section will show how to obtain information if a user is an Expert.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to check if a user is an expert. The namespaces`StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Expert` are described by clicking on respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Expert;
```

Check if a User is an Expert

The simplest way to check if a `User` is an `Expert` is to get an `Expert` based on the user we want to check using the `GetExpert` method of the `ExpertHandler`. If the function returns an `ExpertMember` object, it means that the user is an expert; if it returns `null`, the user is not an expert, as no expert with that unique user id exists in the database.

```
// Get the user to check by id
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

bool isExpert = false;

// Try to retrieve an expert
ExpertBase expert = ExpertHandler.GetExpert(user);

// if returned expert object is not null, the user is an expert
if ( expert != null )
  isExpert = true;
```

2.16.5. Add a Question

The purpose of experts existence in the community is simply to answer questions. However, before they can do so, a question has to be asked. In this section, we will show how a user can add a question to an Expert.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to ask a question by a user. The namespaces`StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Expert` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Expert;
```

Add a Question

To add a question, first a `Questioncode` object has to be created. To create it, we need to provide `question header`, question body, the status of the question and the author of the question. Question header and body are strings that define the question; the question status can be one of: New, Assigned, Published, Revoked or Rejected. For new questions, it is best to set the status to New. Question author can be one of: UserAuthor (community User is the author of the question), AnonymousAuthor (community User is the author, but does not want to be identified by other community members) and GuestAuthor (has no underlying user).

After the Question object is created, the question can be saved in the database using theAddQuestion method of the ExpertHandler class.

```
// Get the user to create an author
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// Create the author of the question.
// If the current user is not logged in user (user is null),
// the created author will be a GuestAuthor. Otherwise,
```

```
// a UserAuthor will be created.
IAuthor author = null;
if ( null == user )
  author = new GuestAuthor("Guest");
else
  author = new UserAuthor(user);

// create the question
Question question = new Question("Header", "Body",
  QuestionStatus.New, author);

// Add question to the database - returned Question
// object has ID property set
question = ExpertHandler.AddQuestion(question);
```

2.16.6. Assign a Question

Before an expert can answer a question, it has to be assigned to him. When the question has been assigned, you can use one of several methods in the ExpertHandler class to retrive the question based on its assignments, answered/not answered status etc.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to ask a question by a user. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Assign a Question

To assign a question, an `AssignedQuestion` object has to be created. Objects of this class bind questions to experts, and allow for assigning a forum topic that relates to the question.

```
// Get a question from database
Question question = ExpertHandler.GetQuestion(322);

// Get an expert from database
ExpertBase expert = ExpertHandler.GetExpert(994);

// Create AssignedQuestion
AssignedQuestion asgndQstn = new AssignedQuestion(question, expert);

// Add the assignment to the database;
// the returned AssignedQuestion has ID property set
asgndQstn = ExpertHandler.AddAssignedQuestion(asgndQstn);
```

2.16.7. Answer a Question

When a question has been assigned to an expert, it can be answered. If the Expert module have the `AutoPublish` property set to true, the question status is automatically changed to

`Published` if it was in `New` or `Assigned` state. Below you can find a short example of how to add an answer to a question.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to answer a question. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Answer a Question

To answer a question, an `Answer` object has to be created. `Answer` object has a reference to the `AssignedQuestion` object, therefore both the question and the expert answering the question are always known. When the `Answer` object has been created, it has to be committed to database using the `AddAnswer` method of the `ExpertHandler` class.

```
// Get an assinged question from database
AssignedQuestion asgndQstn = ExpertHandler.GetAssignedQuestion(987);

// Create an answer, initially in "not approved" state
Answer answer = new Answer("Answer header", "The answer itself",
  AnswerStatus.NotApproved, asgndQstn);

// Add answer to the database;
// the returned Answer object has its ID property set
answer = ExpertHandler.AddAnswer(answer);
```

2.16.8. Approve an Answer

Before publishing the answer to the public, it might be necessary to review the answer to approve it. This section provides information on how to approve an answer.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to answer a question. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Approve an Answer

To approve an answer, the `Answer` object has to be retrieved from the database. Approving an answer means setting the `Status` property to `AnswerStatus.Approved`. When the `Answer` object is changed, it has to be committed to database using the `UpdateAnswer` method of the `ExpertHandler` class.

```
// Get the answer from the database
Answer answer = (Answer)ExpertHandler.GetAnswer(396).Clone();

// Update answer status
answer.Status = AnswerStatus.Approved;

// Update answer in the database
answer = ExpertHandler.UpdateAnswer(answer);
```

2.16.9. Get Questions Assigned to an Expert

Normally, when an expert logs into the site, you want to show the questions assigned. This section explains how to do it in detail.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to answer a question. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Get Questions Assigned to an Expert

To retrieve questions assigned to a specific `Expert`, all we need is to know the `Expert` (e.g. knowing the unique GUID). After that we get the questions assigned to the `Expert` by calling the `GetAssignedQuestions` method of the `ExpertHandler` class – the method then returns the `AssignedQuestionCollection` object, and each of the collection elements (that is, `AssignedQuestion` objects), has a `Question` property which can be used to get the `Question` object itself.

```
// Get the expert
ExpertBase expert = ExpertHandler.GetExpert(333);

// Get expert assigned questions (first page of 100 questions)
AssignedQuestionCollection asgndQuestions =
ExpertHandler.GetAssignedQuestions(expert, 1, 100);

// Get questions itself
QuestionCollection questions = new QuestionCollection();
Foreach( AssignedQuestion aq in asgndQuestions )
{
  questions.Add( aq.Question );
}
```

2.16.10. Get Question Answers

When a user choses a question on the site similar to the question it has, the user might want to see the answers to this question. This section shows how to retrieve answers given to specific questions.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to answer a question. The namespace `StarCommunity.Modules.Expert` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Expert;
```

Get Answers to a Question

The most common situation is when it is needed to display all answers given to a question. The example below shows how one can do that.

```
// Get the question by its ID
Question question = ExpertHandler.GetQuestion(234);

// Get the answers given to this question (first page with 100
// answers)
AnswerCollection answers = ExpertHandler.GetAnswers(question, 1,
100);
```

Get Answers Given by an Expert

It is sometimes usefull to know all answers given by a specific `Expert`, regardless of a `Question`. The code below presents how to do that.

```
// Get the expert by its ID
ExpertBase expert = ExpertHandler.GetExpert(332);

// Get the answers given by this expert (first page with 100
// answers)
AnswerCollection answers = ExpertHandler.GetAnswers(expert, 1, 100);
```

Get an Answer Given by an Expert to a Question

To get only the answer that an `Expert` submitted to a specific `Question`, you can use code similar to the one presented below.

```
// Get the expert by its ID
ExpertBase expert = ExpertHandler.GetExpert(332);

// Get the question by its ID
Question question = ExpertHandler.GetQuestion(323);

// Get the answer given by this expert to this question
Answer answer = ExpertHandler.GetAnswer(question, expert);
```

2.17. Forum

Forums are stored in a tree structure; in the root are the forum instances, followed by discussion rooms, and their child rooms. A forum instance is a way of shielding different rooms from interacting with each other. In a room, different topics can be posted and replied on.

The following kinds are available:

TYPE	DESCRIPTION
Prioritized	A prioritized topic is displayed above regular topics, keeping them there even if they are no longer having active discussions.
Announcement	Announced topics are displayed in all rooms of a forum instance and above prioritized and regular topics.
Locked	A topic can be locked in combination with being announced or prioritized. Locked topics can no longer be replied to.

2.17.1. Adding a Forum

If you want to add a new “forum instance” to the root of the tree this is how you would proceed.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a forum instance. The namespace `StarCommunity.Modules.Forum`is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Forum;
```

Adding the Forum Instance

To add a forum instance we simply create a new `ForumHandler` object instance, together with a new `Forum` object instance. The forum will need a `Site` as a parameter to the constructor; in this case we have selected to supply the `CurrentSite` property, which returns the site we are currently browsing.

```
Forum forum = new Forum(StarSuite.Core.SiteHandler.CurrentSite,
                        "Test Forum");

forum = ForumHandler.AddForum(forum);
```

2.17.2. Adding a Topic

Topics are added to a room and can be created by guests, registered users or users wishing to be anonymous. This article will describe how a registred user creates a new topic.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a topic. The namespaces `StarCommunity.Modules.Forum`, StarCommunity.Core.Modules.Security and `StarCommunity.Core` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunirt.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Forum;
```

Adding the Topic

To add the topic we need a `User` object instance of the author of the topic together with a `Room` object instance of where we want to post the topic. We then construct a `Topic` and supply it to the `AddTopic` method of the `ForumHandler` to store it in the database.

```
//Get the topic author by id
IUser user = (IUser)StarCommunitySystem.
      CurrentContext.DefaultSecurity.GetUser(1234);

//Get the targeted room by id
RoomBase room = ForumHandler.GetRoom(1234);

Topic topic = new Topic(new UserAuthor(user),
                        "Topic subject",
                        "Topic text", room);

topic = ForumHandler.AddTopic(topic);
```

2.17.3. Locking a Topic

By locking a topic you can mark it as not allowing further replies. In this article we will describe how to update a topic to a locked state.

Import Necessary Namespaces

First import the necessary namespaces that will be used to lock a topic. The namespace `StarCommunity.Modules.Forum`is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Forum;
```

Locking the Topic

To lock a topic we first get the `Topic` object instance by its `id`. By changing the `Locked` property we have made the necessary changes in memory. To finalize the locking of the topic we need to store our changes in the database, we do that by calling the `UpdateTopic` method of the `ForumHandler`. The topic is now marked as locked and the appropriate measures can be taken in the user interface of the web page.

```
Topic topic = (Topic)handler.GetTopic(1234).Clone();
topic.Locked = true;

ForumHandler.UpdateTopic(topic);
```

2.17.4. Removing a Topic

This article will describe how to remove a topic from a room.

Import Necessary Namespaces

First import the necessary namespaces that will be used to remove a topic. The namespace `StarCommunity.Modules.Forum`is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Forum;
```

Removing the Topic

To remove a topic we start with getting the `Topic` object instance we want to remove, by its `id`. When we have the object instance we pass it to the `RemoveTopic` method, removing it from the database.

```
Topic topic = handler.GetTopic(1234);

ForumHandler.RemoveTopic(topic);
```

2.17.5. Moving a Topic

Moving of topics can be done in two ways. One way is to change the topics connection to a room, keeping its id and leaving no trace of the move. Another way is by creating a copy in a new room, leaving a trace of the move in its old location, where the trace keeps the old topic id. How this is done depends of the room's `TraceMove` property.

Import Necessary Namespaces

First import the necessary namespaces that will be used to move a topic. The namespace `StarCommunity.Modules.Forum`is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Forum;
```

Moving the Topic

To move the topic we need the object instance of the destination room, the topic to be moved and a `TopicTrace` to leave in the old location. The `TopicTrace` will be used in case the topic's room has the property `TraceMove` set to `true`.

```
//Get the destination room by id
RoomBase destRoom = ForumHandler.GetRoom(1234);
```

```
//Get the topic by id
Topic topic = ForumHandler.GetTopic(1234);

//Move the topic
topic = ForumHandler.MoveTopic(topic,
                  destRoom,
                  new TopicTrace("Trace Subject", "Trace Text"));
```

2.17.6. Adding a Reply

Replies are added to topics and can be authored by guests, registered users or users wishing to be anonymous. This article will describe how to add a reply to a topic.

Import Necessary Namespaces

First import the necessary namespaces that will be used to add a reply. The namespaces `StarCommunity.Modules.Forum`, StarCommunity.Core.Modules.Security and `StarCommunity.Core` are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Forum;
```

Adding the Reply

To add the reply we need a `User` object instance of the author of the reply together with the `Topic` object instance we wish to reply to. We then construct a `Reply` and supply it to the `AddReply` method of the `ForumHandler` to store it in the database.

```
//Get the reply author by id
IUser user = (IUser)StarCommunitySystem.
      CurrentContext.DefaultSecurity.GetUser(1234);

//Get the topic we are replying to by id
Topic topic = ForumHandler.GetTopic(1234);

Reply reply = new Reply(new UserAuthor(user),
                        "Topic subject",
                        "Topic text", topic);

reply = ForumHandler.AddReply(reply);
```

2.17.7. Removing a Reply

This article will describe how to remove a reply to a topic.

Import Necessary Namespaces

First import the necessary namespaces that will be used to remove a reply. The namespace `StarCommunity.Modules.Forum`is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Forum;
```

Removing the Topic

To remove a reply we start with getting the `Reply` object instance we want to remove, by its id. When we have the object instance we pass it to the `RemoveReply` method, removing it from the database.

```
Reply reply = ForumHandler.GetReply(1234);

ForumHandler.RemoveReply(reply);
```

2.18. Image Gallery

The ImageGallery is a central module in StarCommunity Framework because it is used wherever images are handled in a community system. ImageGallery management is done through the `ImageGalleryHandler` class in the StarCommunity.Modules.ImageGallerynamespace.

2.18.1. Adding an Image Gallery

In many cases, an `ImageGallery` is already provided and there is no need to create one. This is the case with `Blog`, `Calendar`, `Expert`, `Contest` and `MyPage` where an `ImageGallery` is created upon object instantiation of these classes and accessible via the `ImageGallery` property. However, you might want to create a new stand-alone `ImageGallery`. This is done via the `ImageGalleryHandler`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.ImageGallery;
```

First we create an `ImageGallery` object.

```
ImageGallery imageGallery =
                        new ImageGallery( "Name", "Description" );
```

At this point the `imageGallery` object only exists in memory, we commit the object to database by calling the `AddImageGallery` method in the `ImageGalleryHandler` class.

```
imageGallery = ImageGalleryHandler.AddImageGallery( imageGallery );
```

Note that the `AddImageGallery` method returns the committed object with the unique `ID` property set.

2.18.2. Removing an Image Gallery

To remove an `ImageGallery` you call the `RemoveImageGallery` method in the `ImageGalleryHandler` class.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery`is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.ImageGallery;
```

First we need the `ImageGallery` to be removed:

```
imageGallery = ImageGalleryHandler.GetImageGallery( 1234 );
```

Then we use the `ImageGalleryHandler` to remove it:

```
ImageGalleryHandler.RemoveImageGallery( imageGallery );
```

2.18.3. Adding an Image

As mentioned in Adding an Image Gallery, there are several classes already providing an `ImageGallery` accessible through the `ImageGallery` property. However, in this sample we use an existing stand-alone `ImageGallery` to put our images in.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries and users. The namespaces `StarCommunity.Modules.ImageGallery`, `StarCommunity.Core and` StarCommunity.Core.Modules.Security are described by clicking on their names. We also need to import `System.IO` for a file `Stream`. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using System.IO;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.ImageGallery;
```

We get the `ImageGallery` via the `ImageGalleryHandler`:

```
imageGallery = ImageGalleryHandler.GetImageGallery( 1234 );
```

Then we create an `Image` object providing the `ImageGallery` and a `System.IO.Stream` object for the image file. We also provide the `imageGallery` where we want the `Image`, the publish state and the current user who is uploading the image.

```
//Get the uploading user
IUser user =
(IUser)StarCommunitySystem.CurrentContext.DefaultSecurity.GetUser(12
34);

//Get the file stream
FileStream stream = new FileStream("foo.jpg", FileMode.Open);

Image image =
```

```
            new Image("foo.jpg", "My first image", stream,
                                  imageGallery,
PublishState.Published, user, false);
```

Note that the image object at this point only exists in memory, and we need to call the `AddImage` method in the `ImageGalleryHandler` to commit it:

```
image = ImageGalleryHandler.AddImage( image );
```

Note that the `AddImage` method returns the committed object with the unique `ID` property set.

2.18.4. Removing an Image

Removing an existing image is done via the `ImageGalleryHandler`.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.ImageGallery;
```

First we get the `Image` object to remove:

```
Image image = ImageGalleryHandler.GetImage( 1234 );
```

Then we remove it:

```
ImageGalleryHandler.RemoveImage( image );
```

2.18.5. Crop and Rotate an Image

The image `Crop` and `Rotate90` methods are located in the `ImageActionHandler` class in the StarCommunity.Modules.ImageGallery.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.ImageGallery;
```

First we need to get an `Image` to edit. This is done using the `GetImage` method in the `ImageGalleryHandler` class. At the same time we create an instance of the `ImageActionHandler` to access the crop and rotate methods:

```
Image image = ImageGalleryHandler.GetImage( 1234 );
```

To begin with, we need a temporary file to save our image during editing, before we commit the changes to database. This can for example be done by using the `System.IO.Path.GetTempFileName()`, which creates such a file for us:

```
string tmpImageFileName = System.IO.Path.GetTempFileName();
```

Then we rotate the image 90 degrees clockwise by calling the `Rotate90` method in the `ImageActionHandler`:

```
ImageAction rotateAction =
      iah.RotateImage90(image.AbsolutePath, tmpImageFileName, true);
```

To crop the image we need to specify the coordinates for the upper-left corner and width and height. These are then passed to the Crop method in the `ImageActionHandler` class:

```
int x = 23;
int y = 54;
int w = 100;
int h = 100;

ImageAction cropAction =
      ImageActionHandler.Crop(image.AbsolutePath, tmpImageFileName,
      image.Width, image.Height, x, y, w, h);
```

The `Rotate90` and `Crop` methods are returning an `ImageAction` object containing data on the changes that were made on the temporary image file. We collect the `ImageAction` objects to an `ImageAction` array:

```
ImageAction[] ia = new ImageAction[] { rotateAction, cropAction };
```

To make the changes to the original `Image` object and commit the changes to database we call the `ImportEditedImage` method in the `ImageGalleryHandler` class providing the image action array:

```
ImageGalleryHandler.ImportEditedImage(image, tmpImageFileName, ia);
```

2.18.6. Getting a Thumbnail of an Image

In most cases you don't want to display an image in the original format. Therefore, we use the `GetThumbnail` method in the `ImageGalleryHandler` where we can define height and width of the returned image. We can then use the `Url` property on the `Thumbnail` to display it on the web page.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.ImageGallery;
```

First we need the `Image` from where we extract the `Thumbnail`. This is done via the `ImageGalleryHandler`:

```
Image image = ImageGalleryHandler.GetImage( 1234 );
```

Now we use the `GetThumbnail` method in the Image class to get a `Thumbnail` where we specify the height and width.

```
Thumbnail thumb =
      image.GetThumbnail(100, 100, ThumbnailFormat.Proportional);
```

The available `ThumbnailFormat` properties are `Proportional`, `ExactandReduceAndCrop`. `ThumbnailFormat.Proportional` scales the image so that its proportions are intact. Therefore you can not expect to get an image size of 100 by 100. `ThumbnailFormat.Exact`stretches the image if necessary to an image size of 100 by 100. `ThumbnailFormat.ReduceAndCrop` makes sure you get a 100 by 100 image by cropping it if necessary.

We may now use the `Url` property in the `Thumbnail` class to display the image on a web page.

2.18.7. Getting Images in an Image Gallery

To get images from an `ImageGallery` to a `ImageCollection` we use the `GetImages` method in the `ImageGallery` class.

Import Necessary Namespaces

First import the necessary namespaces that will be used to manage image galleries. The namespace `StarCommunity.Modules.ImageGallery` is described by clicking on its name. Make sure you add the assemblies as a reference, mentioned in Setting up Visual Studio.

```
using StarCommunity.Modules.ImageGallery;
```

First we get the `ImageGallery` that contains our images. This is done via the `ImageGalleryHandler` class:

```
ImageGallery imageGallery = ImageGalleryHandler.GetImageGallery(
1234 );
```

We get the images in the image gallery `imageGallery`. We take page 1 with 20 items per page sorting on image name ascending:

```
int totalHits = 0;
ImageCollection ic =
                  imageGallery.GetImages(1, 20, out totalHits,
                  new ImageSortOrder(ImageSortField.Order,
                  SortDirection.Ascending));
```

2.19. Moblog

The Moblog module allows MMS messages from mobile phones to be sent to a community running StarCommunity. The Moblog module can receive and store text, image, sound and video content and has a series of configuration options. By defining destination filters in the StarCommunity administration interface content can be stored in the following places:

DESTINATION	DESCRIPTION
MyPage	The content is stored in the MyPage Image Gallery/Document Archive. The correct MyPage is found by matching the MMS sender's phone number against the "msisdn" attribute of the MyPage owner. The attribute to look to can be changed in the Moblog config file.
Selected	The content is stored in an Image Gallery/Document Archive selected by the administrator.
Ignore	The content of a specific type is ignored.

2.19.1. Redirecting an Unwire MMS to a Specific Destination

The default installation of the Moblog module comes integrated with Unwire. Unwire is a mobile enabler company that can deliver MMS messages in an easily read XML format. The Moblog module already handles this format. Though in some cases delivering content to other parts of the community than the MyPage may be necessary, let's say to a club matching the name mentioned in the message. This article will explain how you as a developer can build a web page that serves as the receiving point of the Unwire message, how to parse it and then decide its destination before storing it.

Import Necessary Namespaces

First import the necessary namespaces that will be used to redirect an MMS message. The namespaces`StarCommunity.Modules.Moblog`, `StarCommunity.Modules.Moblog.ContentProviders.Unwire` and `StarCommunity.Modules.ImageGallery`, are described by clicking on their respective names. Make sure you add the assemblies as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Moblog;
using StarCommunity.Modules.Moblog.ContentProviders.Unwire;
using StarCommunity.Modules.ImageGallery;
```

Redirecting the Message

The aim of this web page is to first get the `UnwireContentProvider` singleton and then accept the Unwire request for the web page, mentioned in the Unwire documentation. Parse the XML with the `ParseMmsXml` method to retrieve an `Mms` in-memory object instance. The following code is up to the implementer, but in this case we create an in-memory filter directing images to a specific `ImageGallery`, ignoring all other content types, except text, which is always stored in the message itself. When we finally call the `OnMessageReceived` method the message is stored in the database and all images in it will end up in the `ImageGallery` we specified in the filter.

```
UnwireContentProvider unwireCp = null;
```

```
foreach(ContentProviderBase cp in
                  MoblogModule.Instance.ContentProviders)
{
          if(cp is UnwireContentProvider)
                    unwireCp = (UnwireContentProvider)cp;
}

if(unwireCp == null)
          throw  new  ApplicationException("Unwire  Content  Provider
is
                                              not installed", null);

Response.Clear();
Response.ClearHeaders();
Response.ContentType = "text/plain";
Response.AddHeader("cmd", "asynch-no-trace");
Response.Flush();

System.Xml.XmlDataDocument xmlDoc =
                        new System.Xml.XmlDataDocument();
xmlDoc.Load(Request.InputStream);

Mms mms = unwireCp.ParseMmsXml(xmlDoc);

//Analyze the mms and create a destination filter
MoblogHandler moblogHandler = new MoblogHandler();

MmsDestinationFilter filter =
      new MmsDestinationFilter(mms.ShortCode, mms.MediaCode, null,
                               MmsContentDestination.Selected,
                              ImageGalleryHandler.GetImageGallery(12
                              34),      MmsContentDestination.Ignore,
                              null,     MmsContentDestination.Ignore,
                              null);

//Store the message content in the destinations
//referred to in the filter
unwireCp.OnMessageReceived(mms, filter);
```

2.20. MyPage

The StarCommunity MyPage module contains the functionality that is used on a user's profile.

2.20.1. Blocking a User

Blocking a user, flags that user as blocked in the system. This can then be used to stop a user from communicating with the user that added the block from sending messages, adding as friend, etc.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to block a user. The namespaces`StarSuite.Core`, `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.MyPage`are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Core;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.MyPage;
```

Blocking a User

The code below shows how a user is blocked. Initially, get the two users. Then, get the `MyPage` class for the user who is making the block. The block is created by taking the parameters of this `MyPage` class and the `User` object of the one to block.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// Get the user to block
IUser user2 = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

// Get the my page object for the current user and current site
MyPage mp = MyPageHandler.GetMyPage(user, SiteHandler.CurrentSite);

//Block this user
MyPageHandler.AddBlock(mp, user2);
```

2.20.2. Seeing if a User is Blocked

Since a flag is set to block a user, this flag can then be retrieved to determine wheter or not this user is blocked.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to see if the user is blocked. The namespaces `StarSuite.Core`, `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.MyPage` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Core;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.MyPage;
```

Seeing if a User is blocked

The code section below displays how the `IsBlocked` function is used to see if the user is blocked or not. This function returns `true` if blocked, and `false` if not blocked.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// Get the user to see their block status
IUser user2 = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

// Get the my page object for the current user and current site
MyPage mp = MyPageHandler.GetMyPage(user, SiteHandler.CurrentSite);

//Determine if the user is blocked
bool isBlocked = MyPageHandler.IsBlocked(mp, m_user2);
```

2.20.3. Getting Blocked Users

If you want to display all users a person has blocked as a list on their profile, this can be retrieved from the system.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to get blocked users. The namespaces `StarSuite.Core`, `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.MyPage` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Core;
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
```

```
using StarCommunity.Modules.MyPage;
```

Getting Blocked Users

The code below displays how to get a list of blocked users from a `MyPage` object.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// Get the my page object for the current user and current site
MyPage mp = MyPageHandler.GetMyPage(user, SiteHandler.CurrentSite);


// The list of users that are blocked for this MyPage owner
StarSuite.Security.UserCollection blockedUsers =
                                      mp.GetBlockedUsers(1, 10);
```

2.20.4. Setting a Portrait Image

A portrait image is created on the user by setting the `Portrait` property of the `MyPage` object. This property is an `ImageGalleryImage`.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to set the portrait image. The namespaces`StarSuite.Core`, `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.MyPage`are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using System.IO;
using StarSuite.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.MyPage;
using StarCommunity.Modules.ImageGallery;
```

Setting a Portrait Image

View the code below to see a sample of how the profile portrait image can be set.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

// Get the my page object for the current user and current site
MyPage mp = (MyPage)MyPageHandler.GetMyPage(user,
SiteHandler.CurrentSite).Clone();
```

```
private string exampleFile = @".\Image.jpg";
FileStream fs = new FileStream(exampleFile, FileMode.Open);

using(fs)
{

      //Create the Image object
      Image portrait = newImage("Portrait",
                                    "Some description", fs, user);

      //Set the Portrait property
      mp.Portrait = portrait;

      //Update the MyPage class
      MyPageHandler.UpdateMyPage(mp);
}
```

2.21. NML

NML is a markup language much like HTML. To let community members use NML for producing rich text instead of letting them supply HTML directly has some great advantages.

First of all, you can easily limit what can by done by only defining NML tags that do sane, safe things. Secondly, the community member can not alter the look of the entire page (unless such tags are defined), only the area that the NML output is visible in will be affected. That is, even if opened tags are not properly closed, no “leakage” occurs. Defining Tags in Configuration File.

Example Configuration

This is part of the default configuration. This will be helpful to have available as a reference when reading the explanation of the NML configuration file.

```
<?xml version="1.0" encoding="UTF-8"?>
<NMLSettings
xmlns="http://netstar.se/StarCommunity/NML/NMLSettings.xsd"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://netstar.se/StarCommunity/NML/NMLSettings.
xsd">
<Category Name="general">
<Tag Trigger="font">
<PreTemplate>&lt;span style="{face}{size}"&gt;</PreTemplate>
<PostTemplate>&lt;/span&gt;</PostTemplate>
<Attribute Name="face" DefaultValue="verdana"
                                Template="font-family:{value};">
<AllowedValue>verdana</AllowedValue>
<AllowedValue>arial</AllowedValue>
<AllowedValue>courier</AllowedValue>
</Attribute>
<Attribute Name="size" DefaultValue="10"
Template="font-size:{value}px;">
<AllowedValue>18</AllowedValue>
<AllowedPattern>^1[0-4]$</AllowedPattern>
</Attribute>
</Tag>
</Category>
</NMLSettings>
```

The `Category` tag

The configuration file contains one or more `Category` tags. Any tags defined are under one of these categories. This means that different categories can have completely different tags defined, or different implementations of the same tag.

When rendering NML, the category to use can be specified, otherwise the category named “general” is used.

The `Tag` tag

This tag has the mandatory `Trigger` attribute which is the name that defines the tag. If you want to be able to use “[b]foo[/b]”, the `Trigger` value would be “b”.

The `PreTemplate` tag

This tag holds the template text for the output before the text that the tag encloses. If you have “[b]foo[/b]”, this is the template for the text that should be added before “foo” in the output.

The `PostTemplate` tag

This tag holds the template text for the output after the text that the tag encloses. If you have “[b]foo[/b]”, this is the template for the text that should be added after “foo” in the output.

The `Attribute` tag

There can be zero or more `Attribute` tags for each `Tag` tag. These define possible attributes to that NML tag. The mandatory `Name` attribute defines the name of the NML attribute. This is used as the key when inserting the attribute value in the `PreTemplate` or `PostTemplate` text. In the template texts “{keyname}” is replaced with the attribute value with that name. The mandatory `DefaultValue` attribute defines the default value of the NML attribute. The mandatory `Template` attribute defines the template text for the attribute in the output. Each `Attribute` tag may have zero or more `AllowedValue` or `AllowedPattern` tags, these are used to verify that the user-provided value for the attribute is sane. `AllowedValue` defines a static string to validate against, `AllowedPattern` defines a regular expression to use for validation. If any of the `AllowedValue` or `AllowedPattern`s match the supplied value for the attribute, the input will be accepted. Otherwise the supplied value will be ignored.

2.21.1. Rendering NML Content

The main purpose of the NML module is to render NML code. This is very easy to do once you have decided on what tags to use and when you have added them in the configuration file.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed for this. The namespace `StarCommunity.Modules.NML` is described by clicking on the name. Make sure you also add the mentioned assembly as a reference, as mentioned in section 1.1.1.

```
using StarCommunity.Modules.NML;
```

Limiting Maximum Word Length

To render an NML string, simply call the static `Render` method in the `NMLModule`. It is also possible to provide an NML category as an additional argument, if desired.

```
NMLModule.Render("a [b]string[/b]");
```

2.21.2. Limiting Maximum Word Lengths

There is functionality in the NML module to encode text with HTML, render NML and limit max word length, all in one single step. This method is helpful in many cases where you want to render NML, and don't want to do any additional manipulation of the text manually.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed. The namespace `StarCommunity.Modules.NML`is described by clicking on the name. Make sure you also add the mentioned assembly as a reference, as mentioned in section 1.1.1.

```
using StarCommunity.Modules.NML;
```

Limiting Maximum Word Length

To have a NML string rendered, with words longer than a given length broken up, simply call the static `Format` method in `NMLModule`. It is also possible to provide an NML category as an additional argument, if desired.

```
NMLModule.Format
      ("a string with a [b]looooooooooooooooooong[/b] word", 7);
```

2.22. OnlineStatus

The StarCommunity OnlineStatus module provides the tools to see which users are logged in to the site. Functionality includes seeing if a user is currently online, getting the date that a user was last online and getting a list of the last logged in users.

2.22.1. Seeing if a User is Online

It is easy to see if a user is currently online. This can be used on the site to indicate the user's status.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to see if a user is online. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.OnlineStatus` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.OnlineStatus;
```

Seeing if a User is Online

The following code shows how to retrieve a user and to determine the online status.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);

//Flag to see if the user is online
bool isOnline = OnlineStatusModule.IsOnline(user);
```

2.22.2. Getting a User's Last Login Date

The OnlineStatus module provides functionality to return a date indicating the date and time that a user was last online. This date can then for example be displayed on the user's profile page.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to see if a user is online. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.OnlineStatus` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.OnlineStatus;
```

Getting a User's Last Login Date

The following code displays how to retrieve the last date a user was logged in.

```
//Get the logged in user
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.GetUser(1234);


//Get the date this user was last online
DateTime lastLogin = OnlineStatusModule.GetLastLogin(user);
```

2.22.3. Getting Currently Logged in Users

Most communities want to display a listing of the last users that logged it. This is easy to do using the OnlineStatus module.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to see if a user is online. The namespaces, `StarSuite.Core`.Modules.Security and `StarCommunity.Modules.OnlineStatus` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Security;
using StarCommunity.Modules.OnlineStatus;
```

Getting Currently Logged in Users

The code below shows how to get the last ten logged in users.

```
int numUsersToGet = 10;

UserCollection onlineUsers =
             OnlineStatusModule.GetLastLogins(numUsersToGet);
```

2.23. Poll

The Poll module provides functionality that enables the creation of polls, voting and getting the vote count of specific choices within a poll. Voting in polls is available not only for the community members but community site visitors can also take part in voting; this can be useful to get feedback even from people who are not members.

2.23.1. Adding a Poll

Before a poll can be used, first it has to be created and added to the community. To add a poll, first an object of `Poll` type has to be created, and then it has to be committed in the database using the `AddPoll` method of the `PollHandler` class.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add a poll. The namespaces `StarSuite.Core`, `StarCommunity.Core` and `StarCommunity.Modules.Poll`are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarSuite.Core;
using StarCommunity.Core;
using StarCommunity.Core.Modules;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Poll;
```

Add a Poll

In this example, we will add a test poll to the community. The poll will contain three choices, and will not be assigned to any site. First, an object of the `Poll` type is created, and then it has to be committed to the database using the `AddPoll` method of the `PollHandler` class.

```
// Poll text
string text = "Best hamburgers are made by:";

// Poll author - current user will be the author
IUser currentUser = (IUser)StarCommunitySystem.
              CurrentContext.DefaultSecurity.CurrentUser;
IAuthor author = new UserAuthor(currentUser);

// Poll activity
bool isActive = true;

// Poll site - when set to null, the poll is not assigned
ISite site = null;

// Start and end date when voting can occur
DateTime start = new DateTime(2007,3,1);
DateTime end = new DateTime(2007,4,1);

// Create a poll object
```

```
Poll poll = new Poll(text, author, isActive, site, start, end);

// Create two choices – when creating Choice set text and order
Choice choice1 = new Choice("McDonald's", 0);
Choice choice2 = new Choice("Burger King", 1);

// Add choices to the poll
poll.Choices.Add( choice1 );
poll.Choices.Add( choice2 );

// Commit poll object in database;
// after this operation poll's ID property is set
poll = PollHandler.AddPoll(poll);
```

2.23.2. Removing a Poll

When a poll is no longer needed, it can be removed from the community database. To remove a poll, we need to get the poll information from the database (get the `Poll` object). Then, the poll can be deleted using the `RemovePoll` method of the `PollHandler` class.

Import Necessary Namespaces

First, import the necessary namespace that will be used to remove a poll. The namespace `StarCommunity.Modules.Poll` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Poll;
```

Remove a Poll

In the example below, we assume that we know which poll that shall be removed – the poll `ID` is known (previously selected or found). The `Poll` object is created by getting it from the database by `ID`, and then the poll is removed using the `RemovePoll` method of the `PollHandler` class.

```
// Get the poll from the database
Poll poll = PollHandler.GetPoll(334);

// Remove the poll from the database
PollHandler.RemovePoll(poll);
```

2.23.3. Voting in a Poll

Voting can be available for community members only, or for all site visitors. If you decide to use the `Vote` object constructor with the user parameter, you have to remember that logged in users will be able to vote only once in a poll, because the framework checks whether or not a user already voted in the poll. This allows for controlling the voting process – one user cannot vote more than once. On the other hand, you may decide not to register who voted and

how they voted, and not to provide user information to the vote. This way un-registered users are allowed to vote, users can vote more than once.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to vote in a poll. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Poll` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Poll;
```

Voting in a Poll

To vote in a poll, you have to get the `Poll` object that the user wants to vote in. After that, the `Choice` within the `Poll` has to be selected, and used to construct the `Vote` object. In this example the `Vote` object is created using a constructor that takes the current user as the one that votes in the poll.

```
// Get the poll to vote
Poll poll = PollHandler.GetPoll(11);

// Get the choice to vote to
Choice choice = poll.Choices[1];

// Get the current user that will vote
IUser currentUser = (IUser)StarCommunitySystem.
CurrentContext.DefaultSecurity.CurrentUser;

// Create Vote object
Vote vote = new Vote(choice, currentUser);

// Register the vote in the database
PollHandler.Vote(vote);
```

2.23.4. Display the Current State of a Poll

When creating a poll, there is a need not only to allow users to vote in it, but also to show them the results of the voting. There are two main object properties that help to determine current status: `Poll.VoteCount`, which gets the total number of votes in this poll, and `Choice.VoteCount`, which gets the number of votes of this choice. Knowing those two numbers, it is easy to calculate distribution of votes between choices. In this example we will create a table with percentage values that describe vote distribution in the poll.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to vote in a poll. The namespaces `StarCommunity.Core` and `StarCommunity.Modules.Poll` are described by clicking on

their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Poll;
```

Get the Current State

In the example below, we will create a table of integer values that represent percentage distribution of votes among poll choices. In a real life scenario, these numbers can be used e.g. to display a percentage bar chart that will show how people voted in this poll.

```
// Get the poll get state
Poll poll = PollHandler.GetPoll(11);

// Create table to keep the percentages
int[] percentage = new int[poll.Choices.Count];

int choiceVoteCount = 0;
int totalVotes = poll.VotesCount;

// Fill the table with calculated percentages
for( int ix = 0; ix < poll.Choices.Count; ix++ )
{
  choiceVoteCount = poll.Choices[ix].VoteCount;
  percentage[ix] = (int)((((float)choiceVoteCount)/totalVotes)*100);
}
```

After a poll ha been retrieved from the database, we have access to all the properties of the poll itself, as well as its choices. There is no need to use other `PollHandler` methods to retrieve information about the poll – the properties provide all necessary data.

2.23.5. Adding Choices after Creation

The poll choices do not have to be added before the poll is committed in the database. It is possible to create a poll with its text (a question), and after that to add choices (e.g. after a research on what choices are available). This example will present how to update a poll with new choices.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add choices to a poll. The namespace`StarCommunity.Modules.Poll` is described by clicking on its name. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Modules.Poll;
```

2.23.6. Add Choices to Existing Poll

First, we will need to retrieve the `Poll` object from the database. After that, we can create as many `Choice` objects as we need and add them to the `Choices` collection of the `Poll` object. When all `Choice` objects have been added, we need to save the changes in the database – we use the `UpdatePoll` method of the `PollHandler` class. The `PollHandler` adds, updates or removes any changed choices within the `Poll`.

```
// Retrieve Poll from the database
Poll poll = (Poll)PollHandler.GetPoll(33).Clone();

// Determine current maximum choice order
int maxOrder = 0;
foreach( Choice ch in poll.Choices )
{
  if ( ch.Order > maxOrder )
    maxOrder = ch.Order;
}

// Create two additional choices to add - when creating Choice
// set text and order
Choice choice1 = new Choice("Added choice 1", ++maxOrder);
Choice choice2 = new Choice("Added choice 2", ++maxOrder);

// Add choices to the poll
poll.Choices.Add( choice1 );
poll.Choices.Add( choice2 );

// Commit poll object in database
poll = PollHandler.UpdatePoll(poll);
```

2.24. StarViral

The StarViral Module provides a useful viral marketing tool to attract more members to the community. We can imagine a scenario when each member of the community is granted points for each new member that is added from the user's referral – this scenario is very easy to implement using the StarViral Module. Campaigns can be administered using the administration interface. Creation of referrals does not have to be done within a campaign, but it is very useful to do so. When referrals are categorized within campaigns, it is easy to compare the number of new members attracted and the number of referrals created depending on the campaign rules, which helps to plan the next campaigns.

2.24.1. Adding a Referral

Adding a referral will store a record between the two users. If the referred user registers with her/his e-mail address, the referral record will be updated and reflect the successful referral. A referral does not have to belong to a campaign, but preferably it should, making it possible to see the results in the administration interface.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to add a referral. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.StarViral`are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.StarViral;
```

Adding a Referral

The code below will create a referral record between the logged in user and their friend.

```
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

Campaign campaign = StarViralHandler.GetCampaign(1);

Referral re = new Referral("test@abc.com", "Test user name", user,
campaign);
re = StarViralHandler.AddReferral(re);
```

2.24.2. Display the State of Referrals

The referrals a user has made can be retrieved and displayed as a listing with the registration state or as a count on their profile page.

Import Necessary Namespaces

First, import the necessary namespaces that will be used to display the state of referrals. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.StarViral` are described by clicking on their respective names. Make sure you add the assembly as a reference, mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.StarViral;
```

Display the State of Referrals

The referrals can be retrieved easily by using the code below. If the boolean property `HasRegistered` is set to `true`, the referred user has registered, if `false`, the user has not registered.

```
IUser user = (IUser)StarCommunitySystem.
                  CurrentContext.DefaultSecurity.CurrentUser;

//Get all referrals a user has made to display in a listing
int totalHits = 0;
ReferralCollection referrals = StarViralHandler.GetReferrals(user,
1, 10, out totalHits,
new ReferralSortOrder(ReferralSortField.ReferralDate,
SortDirection.Descending)
);


//Get the total number of referrals a user has made
int myReferralCount = StarViralHandler.GetNumberOfReferrals(user,
false);
```

2.25. Webmail

Webmail management in StarCommunity is done through the `WebmailHandler` class in the `StarCommunity.Webmail` namespace.This article shows you, the developer, examples of how to add Webmail functionality to a community site with the help of the StarCommunity Framework.

2.25.1. Getting the status of an account

It is essential to know whether a user has a mail account, and if so, that it is active.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Webmail` are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Getting the Account for a User and the Status of that Account

First we get the desired `User` object, then the `MailAccount` for that user, and from the account, we can then get the `MailAccountStatus`.

```
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
MailAccountStatus status =
WebmailHandler.GetMailAccountStatus(account);
```

`MailAccountStatus` itself is an enum with the values `Active`, `Deactivated` and `DoesNotExist`. This way you can tell if there is an account, and if so, if it is active or not.

2.25.2. Creating an account

For a site member to be able to receive e-mail, they need to have an account as well as an address associated with that account. If a user does not already have an account, one will need to be created.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Webmail` are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Creating the Account

To create the account, we get the `User` for which the account is to be created, and then call the `AddMailAccount` method in the `WebmailHandler`, specifying the local part of the address (the part before the "@" sign) to be associated with the new account. Optionally you can also supply a `Domain`. If no domain is supplied, the one set in the configuration file is used as the default.

```
Domain domain = WebmailHandler.GetDomainByDomainName("my.domain");
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
WebmailHandler.AddMailAccount(user, "address.part", domain);
```

The account is now created on the mail server, and incoming messages to this address will be accepted from this point on.

It is important to note that the local part of the address has to be unique; it is therefore a good idea to think of some means of making sure that collisions will not occur. For instance, the StarCommunity username may be used (as that is known to be unique), provided that it does not contain any illegal characters.

2.25.3. Disabling, Reactivating and Permanently Removing Accounts

It may be of interest to disable an account for different reasons or to remove an account entirely.

For instance, the mail account may be a premium service, which should be disabled if the user stops paying. Another example may be a misbehaving user that should have their account removed.

Keep in mind that when a StarCommunity user is soft-removed, reactivated or permanently removed, these changes also reflect on the user's mail account automatically, if there is one.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed for this. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Webmail` are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Soft Removing an Account

To disable an account, we get the account for the relevant user and call the `SoftRemoveMailAccount` method in the handler. Note that this is automatically done when soft removing a user in the administration interface.

```
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
WebmailHandler.SoftRemoveMailAccount(account);
```

Now the account has been disabled. This means that no mail can be delivered to it and that you can no longer get mail from this account.

Restoring a soft removed account

To reactivate a disabled account, we get the account for the relevant user and call the `SoftRestoreMailAccount` method in the handler. Note that this is automatically done when restoring a user in the administration interface.

```
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
WebmailHandler.SoftRestoreMailAccount(account);
```

Now the account has been restored. Any messages which were there before the account was disabled are now available again.

Permanently Removing an Account

To permanently remove an account, we get the account for the relevant user and call the `RemoveMailAccount` method in the handler. Note that this is automatically done when permanently removing a user in the administration interface.

```
WebmailHandler handler = new WebmailHandler();
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
WebmailHandler.RemoveMailAccount(account);
```

Now this account has been permanently removed. A new account would have to be added for this user to be able to use Webmail functionality again.

2.25.4. Managing the Mailbox Tree for an Account

The mail server has a tree structure of mailboxes, in which the actual messages reside. You can create and remove folders as you wish using the Webmail Module. Typically it may be of interest to create a “Sent” folder in which copies of outbound messages can be saved.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Webmail` are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Creating a Mailbox

To create a new mailbox, we get the mail account for the user, then the root mailbox which we will use as the base in this example, and then call the `AddMailbox` method to have a new mailbox with the specified name created under the given mailbox.

```
IUser user = (IUser)StarCommunitySystem.

           CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
Mailbox root = account.RootMailbox;
WebmailHandler.AddMailbox("name", root);
```

Accessing a Mailbox

The mailboxes in an account form a tree structure. The root mailbox is the only one which is explicitly referenced from the outside, but it is easy to get to any given mailbox using the `ChildMailboxes` property which is available on all mailboxes.

```
IUser user = (IUser)StarCommunitySystem.
           CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
Mailbox root = account.RootMailbox;
Mailbox subMailbox = root.ChildMailboxes["name"];
```

This way, you can access a mailbox by its name, or you can iterate over the `ChildMailboxes` as a list, depending on your needs.

2.25.5. Getting Messages

The center piece of any Webmail implementation is the messages. This article will show examples of how to access the actual messages.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed for this. The namespaces `StarCommunity.Core`, StarCommunity.Core.Modules.Security and

StarCommunity.Modules.Webmail are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Getting a List of Messages

To get a list of messages, we first get the mail account of the user, we then call the GetMessages method in the WebmailHandler, specifying the mailbox, the paging information (page size and page number to retrieve) and the sort order.

```
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
MessageSortOrder order =
new MessageSortOrder(MessageSortField.Date,
SortDirection.Descending);
Mailbox mbox = account.RootMailbox;
int totalHits = 0;
MessageCollection messages = WebmailHandler.GetMessages(mbox, 1, 20,
out totalHits, order);
```

The returned MessageCollection now holds the first 20 messages from the specified mailbox. This collection can now for example be iterated over to display a message listing, with all the appropriate information available.

Notes Regarding Handling of Received Messages

It is important to note that Message is the base class for ServerMessage (any message that has been retrieved from the mail server) and LocalMessage (any message that is being constructed locally to be sent). Some properties specific to an incoming message is only available when it is treated as a ServerMessage, so a cast may be necessary if the message is for example taken from a MessageCollection.

It is also worth noting that accessing the TextBody, HTMLBody and Attachments properties of a ServerMessage imposes significant overhead compared to other properties, as these require the full message to be transferred from the mail server. Doing so also sets the message as read.

It is therefore not advisable to include these properties in message listings and similar if it can be avoided, but rather only use those when displaying a complete individual message.

2.25.6. Sending a Message

Another very central part of any Webmail implementation is the possibility to send mail. In this article we will show a basic example of how to send mail.

Import Necessary Namespaces

First, import the necessary namespaces that will be needed for this. The namespaces `StarCommunity.Core`, `StarCommunity.Core`.Modules
This namespace contains the interface `IStarCommunityEntity` and the abstract implementation classStarCommunityEntityBase. IStarCommunityEntity implements the blueprint for tagging, attributes, rating and categorization. Also the Author classes and interfaces are located here, allowing for guests and users to identify themselves when making posts.
StarCommunity.Core.Modules.Security and `StarCommunity.Modules.Webmail`are described by clicking on their respective names. Make sure you also add the mentioned assemblies as references, as mentioned in section 1.1.1.

```
using StarCommunity.Core;
using StarCommunity.Core.Modules.Security;
using StarCommunity.Modules.Webmail;
```

Sending a Message

First, we get the mail account of the sending user, we use the account to get the sender's mail address, so that we can set that as the `From` address. We then proceed by setting other properties of the `LocalMessage` object. To finally send the message, we call `SendMessage` in the `WebmailHandler`.

```
IUser user = (IUser)StarCommunitySystem.
          CurrentContext.DefaultSecurity.GetUser(17);
MailAccount account = WebmailHandler.GetMailAccount(user);
LocalMessage message = new LocalMessage();
message.From = wh.GetMailAddress(account);
message.To.Add(new MailAddress("Name", "email@domain"));
message.TextBody = "Text version";
message.HTMLBody = "HTML version";
WebmailHandler.SendMessage(message);
```

Note that it is recommended to either have only `TextBody` set or to have both `TextBody` and `HTMLBody` set, this is for maximum compatibility with different mail user agents.

2.25.7. The Configuration File

The configuration file for the Webmail Module requires custom values for the module to work as it relies on an external mail server.

In this section we will go through the meaning of the different settings.

PARAMETER	TYPE	DESCRIPTION
IMAPBaseFolder	String	Name of the base folder on the IMAP server. Typically “INBOX”.
IMAPFolderSeparator	String	The folder separator on the IMAP server. Typically a single dot (“.”).
IMAPPort	Int	The port number the IMAP server is listening on. Typically 143.
IMAPServer	String	The hostname or IP that the IMAP server is reachable at.
SMTPPort	Int	The port number the SMTP server is listening on. Typically 25.
SMTPServer	String	The hostname or IP that the SMTP server is reachable at.
UsernamePrefix	String	The prefix used for webmail IMAP account usernames. Check with the server administrator.
WebserviceURL	String	The URL for the webservice used for creating accounts, etc. Check with the server administrator.
AccountQuota	Int	Default IMAP account quota. In kilobytes.
WebserviceSecret	String	The shared secret used when communicating with the webservice. Check with the server administrator.
IMAPAuthMode	String	The authentication mode used with the IMAP server. Available are “plain” or “crammd5”.
DomainName	String	The domain name that will be used as the default domain when adding webmail addresses.
DiskCachePath	String	The path to a folder dedicated to storing cached copies of mail data. This folder must be writable to the Windows user that runs the web application.

The configuration file for the Webmail Module requires custom values for the module to work as it relies on an external mail server.

3. Extending StarCommunity

3.1. Extending StarCommunity classes

This tutorial describes howto create a derived class with custom attributes, and that exposes those attributes as fixed properties in the derived class. This is in many cases a more elegant approach than accessing the attributes directly via the attribute name.

In this tutorial we assume that we have a community implementation that requires the `EntryComment` class in `Blog` to have an `ImageGallery` connected to it, where users can upload pictures when commenting an `Entry`. One way of accomplishing this would of course be to just add an attribute to the `EntryComment` as described in section 2.5 and be done with it. However, there is a more elegant way to use attributes without the need of keeping track of the attribute names in your ASP.NET page.

We start by creating a new class derived from EntryComment.

```
using System.Collections.Generic;
using StarCommunity.Modules.Blog;
using StarCommunity.Modules.ImageGallery;

namespaceMyCommunity
{
    //Inherit the EntryComment class
    public class MyEntryComment : EntryComment
    {
        // The constructor takes DbDataReader and passes it to base
class
        public MyEntryComment(DbDataReader reader) : base(reader) { }

        // Define the ImageGallery property for get and set
        public ImageGallery ImageGallery
        {
            get{return
this.getAttributeValue<ImageGallery>("attr_ig");}
set{blog.entity.SetAttributeValue<ImageGallery>("attr_ig",value));}
        }
    }
}
```

In order to make StarCommunity aware of this new type so instances of the type can be returned, we need to create an EntityProvider.This is done by creating a new class,implementing`IEntityProvider` in the `StarSuite.Core.Data` namespace.

```
using System.Collections.Generic;
using StarSuite.Core.Data;
using StarCommunity.Modules.Blog;
using StarCommunity.Modules.ImageGallery;
```

```
namespace MyCommunity
{
    //Implement the IEntityProvider interface
    public class MyEntryCommentEntityProvider :
StarSuite.Core.Data.IEntityProvider
    {
private static MyEntryCommentEntityProvider
m_myEntryCommentEntityProvider =
null;

        //Singleton
        public static StarSuite.Core.Data.IEntityProvider
GetProviderInstance()
        {
            if (m_myEntryCommentEntityProvider == null)
              m_myEntryCommentEntityProvider = new
MyEntryCommentEntityProvider ();

            return m_myEntryCommentEntityProvider;
        }

        // Override the GetEntityInstance by reader method
public object GetEntityInstance(Type type, DbDataReader reader)
        {
            //If the specified type is EntryComment or MyEntryComment,
call
            //MyEntryComment constructor that will just pass the reader
argument to
            //its base class
            if (type == typeof(EntryComment) || type ==
typeOf(MyEntryComment)
                return new MeEntryComment(reader);
        }

        // Override the GetEntityInstance by id method
        public object GetEntityInstance(Type type, int id)
        {
            //If the specified type is EntryComment or MyEntryComment,
just get
            //EntryComment by id via BlogHandler
if (type == typeof(EntryComment) || type == typeOf(MyEntryComment)
                return BlogHandler.GetEntryComment(id);
        }
}
}
```

Now, we have created our entity provider that can return instances of your new type MyEntryComment. Only one more thing remains and that is to register it in the EntityProvider.config, so StarCommunity knows that this entity provider should override the existing one for EntryComments.

```
<EntityProvider>
<Name>MyCommunity.MyEntryCommentEntityProvider, MyCommunity</Name>
```

```
<SupportedTypes>
          <SupportedType>
<Name>MyCommunity.MyEntryProvider, MyCommunity</Name>
</SupportedType>
<SupportedType>
<Name>StarCommunity.Modules.Blog.EntryComment,
StarCommunity.Modules.Blog</Name>
</SupportedType>
</SupportedTypes>
</EntityProvider>
```

All request for the type `StarCommunity.Modules.Blog.EntryComment` and `MyCommunity.MyEntryComment` will now be run through `MyEntryCommentEntityProvider`.

Below is an example where we get the images from the image gallery property in `MyEntryComment` class.

```
using StarCommunity.Modules.Blog;
using MyCommunity;

//Get an entry comments
MyEntryComment comment = (MyEntryComment)BlogHandler.GetEntryComment(1234);
ImageCollection ic =
          comment.ImageGallery.GetImages(1, 20, out totalHits,
          new ImageSortOrder(ImageSortField.Order, SortDirection.Ascending));
```

3.2. Benefit from StarCommunity functionality in third party classes

Third part classes can benefit from StarCommunity functionality by inheriting the StarCommunityEntityBase class or implementing its interfaces. This tutorial describes the creation of a third party class, MyClass, which inherits the `StarCommunityEntityBase` class in the `StarCommunity.Core.Modules`namespace. `StarCommunityEntityBase` requires that a unique id for the type is passed to its constructor and gives inheriting classes the properties and methods for StarCommunity Categories, Tags, Attributes and Rating.

```
using StarCommunity.Core.Modules;

namespace MyCommunity
{
    //Inherit the StarCommunityEntityBase class
    public class MyClass : StarCommunityEntityBase
    {
private string m_name;

        public MyClass(int id, string name) : base(id)
        {
             m_name = name;
}

        public string Name
        {
            get{return m_name;}
            set{m_name = value;}
        }

    }
}
```

In StarCommunity there is a handler for every module, whichcontains the methods, which in turn does the database communication. In this example we create one for MyClass. It will be used in the entity provider (see 1.2.4) to get a MyClass instance from ID.This is necessary for StarCommunity to handle this type properly. However, this particular method may be implemented in any way you wish, just as long as it returns and instance of MyClass based on ID.

```
namespace MyCommunity
{
    public class MyClassHandler
    {
        //In reality, this method would probably involve a database query to create
        //the object from database.
        public static GetMyClass(int id)
        {
            return new MyClass(id, "myclass name");
        }

    }
}
```

Since there aremethods in StarCommunity that returns collections of entities (e.g. `CategorizedEntityCollection`), StarCommunity needs to recognize the new `MyClass` type. We need to create an `EntityProvider` for `MyClass`.

```
using System.Collections.Generic;
using StarSuite.Core.Data;
using MyCommunity;

namespace MyCommunity
{
    //Implement the IEntityProvider interface
    public class MyClassEntityProvider : StarSuite.Core.Data.IEntityProvider
    {
        private static MyClassEntityProviderm_myClassEntityProvider = null;

        //Singleton
        public static StarSuite.Core.Data.IEntityProvider GetProviderInstance()
        {
            if (m_myClassEntityProvider == null)
                m_myClassEntityProvider = new MyClassEntityProvider ();

            return m_myClassEntityProvider;
        }

        // Implement the GetEntityInstance by reader method
        //to construct your object from database
        public object GetEntityInstance(Type type, DbDataReader reader)
        {
            if (type == typeof(MyClass)
                 return new MyClass(reader.GetInt32(0), reader.GetString(1));
        }

        // Implement the GetEntityInstance by id method
        public object GetEntityInstance(Type type, int id)
        {

if (type == typeof(MyClass)
                return MyClassHandler.GetMyClass(id);
        }
    }
}
```

Now we may use the category system to categorize objects of type MyClass

3.2.1.Categorize MyClass entities

Since MyClass is derived from `StarCommunityEntityBase`, you may use the category system directly as you would with any other StarCommunity class. Here we have some examples of an Add and Update method for MyClass, notice that after running the MyClass database-query we call the UpdateEntity method of the base class, which is `StarCommunityFactoryBase`.

```
using StarCommunity.Core.Modules.Data;
using StarCommunity.Core.Modules.Categories;
```

```
using MyCommunity;
....

//Get an instance of MyClass via id
MyClass myClass = MyClassHandler.GetMyClass(1);

//Get a category by id
Category category = CategoryHandler.GetCategory(1);

//Add the category
myClass.Categories.Add(category);
MyClassHandler.UpdateMyClass(myClass);

....

public class MyClassHandler :
        StarCommunity.Core.Modules.Data.StarCommunityFactoryBase
{

// inserting the data
public static void AddMyClass(MyClass mc)
{
bool inTransaction = DatabaseHandler.InTransaction;
if (!inTransaction)
DatabaseHandler.BeginTransaction();

        try
{
int newId = Convert.ToInt32(DatabaseHandler.
GetScalar(true, "spAddMyClass", parameters));

base.UpdateEntity(mc, newId);

if (!inTransaction)
DatabaseHandler.Commit();
}
catch (Exception)
{
if (!inTransaction)
DatabaseHandler.Rollback();

throw;
}
    }

// updating the data
public static void UpdateMyClass(MyClass mc)
{
bool inTransaction = DatabaseHandler.InTransaction;
if (!inTransaction)
DatabaseHandler.BeginTransaction();

        try
{
DatabaseHandler.
               ExecuteNonQuery(true, "spUpdateMyClass",
parameters));
```

```
base.UpdateEntity(mc);

if (!inTransaction)
DatabaseHandler.Commit();
}
catch (Exception)
{
if (!inTransaction)
DatabaseHandler.Rollback();

throw;
}
    }
}
```

UpdateEntity saves all categories, tags, attributes and rating settings on MyClass. UpdateEntity has two overloads, during an insert we call the overload where we can pass the new id, since it's not in our object at the moment, and when we do an update, we simply only pass the "mc" variable. This is important to remember, since when passing an ID some initialization may occur which is unnecessary during an update, so keep track of which overload you call.

3.2.2.Retrieving categories for MyClass

To get a collection of all categories connected to an entity, you just call the Categories property on the categorizable entity MyClass just as for any StarCommunity object.

```
//Get the MyClass to check for categories
MyClass myClass = MyClassHandler.GetMyClass(1);

//Get the categories for the blog
CategoryCollection categoryCollection = myClass.Categories;
```

3.2.3.Retrieving MyClass entities based on categories

Just as for StarCommunity objects, you may retrieve collections of categorized MyClass entities via the category handler.

```
//Get the category for which we want entities
Category category = CategoryHandler.GetCategory(1);

//Add the category to the category collection
CategoryCollection categoryCollection = new CategoryCollection();
categoryCollection.Add(category);

//Get entities of type MyClass that have been categorized with
category
int totalItems = 0;
CategorizableEntityCollection categorizedEntities =
          CategoryHandler.GetCategorizedItems(typeof(MyClass),
          categoryCollection, 1, 10, out totalItems);
```

3.3. Use Netstar Cache system for third party implementations

The Netstar cache system is located in the `StarSuite.Core.Cache` namespace. The main class being the CacheHandler, retrieves and stores caches, keeps track of dependencies and synchronizes events over a webserver cluster. Every object cached in this system can (if chosen to be implemented) be identified by its primary cache key. The cache system is set to detect this primary cache key in any lists or collections it has in its cache tree, hence removing the cache by its primary cache key would also remove the cached lists containing it.

The primary cache key is definied by implementing the `ICacheable` interface and its `CacheKey` property.

```
public class CachedClass : StarSuite.Core.Cache.ICacheable
{
    private int m_id;
    public CachedClass(int id)
{
       m_id = id;
}

    public string[] CacheKey
    {
        get { return new string[] { "tree", "branch", "leaf",
                                    id.ToString() }; }
    }
}
```

Every part added to the key represents going deeper into the tree-structure, which means if “tree”, “branch” would be removed, all leafs under it will go with it.

The following code sample shows how the dependencies are automatically being registered when the containing “co” instance is in the cached list. RemoveCachedObject will also remove the list from the cache.

```
List<CachedClass> list = new List<CachedClass>();

CacheHandler.SetCachedObject(list, “tree”, “branch”, “leaflist”);

CachedClass co = new CachedClass(1234);
List.Add(co);

CacheHandler.RemovedCachedObject(co);
```

Having this is mind and implementing it in a third-party solution can save a lot of time and boost performance of features implemented into the StarCommunity platform.