.Net Business Components

What they are and how they are built

Ver. 1.2

Official Documentation

.Net Business Components - Ver. 1.2

What they are and how they are built

08. Oct. 2009

Official Documentation

Dep. 9 - Department for Information Technology
9.7

Author:

Zeno Moriggl

Content

1. Abstract

This document describes what a .Net Business Component is, how it is structured internally at design time and how it finally is packaged. The audience of this document are in the first place developers who are working on further automation in creation of Business Components and those who want to understand in detail how all the stuff internally works.

Most of the things that you will read about in this document are less important when developing Business Components with the BC Designer, which cares about realizing exactly what is described here (and something more).

2. How to read this document

This document describes the conventions to follow when creating services hosted by the CEE.Net. Each rule is marked by a number enclosed with square brackets, preceded by a letter:

- **O** means that the rule has been introduced due to an organizational guideline;
- **T** means that the rule has a technical background.

The following verbs have a very precise meaning:

- MUST do: you are obliged to do
- MUST NOT do: it is forbidden to do
- SHOULD do: you are not obliged to do but it is recommended to
- SHOULD NOT do: you are not forbidden to do but it is recommended not to do

The term `<root namespace>` stands for the root namespace of your organization. Whenever found in the full name of CEE.Net's data types or in CEE.Net's assembly names, the expression stands for the root namespace of the Autonomous Province of Bolzano.

2.1. What Else Should You Read?

For a complete understanding you should be familiar with the following documents:

- Reference Architecture.pdf
- Application Versioning.pdf

3. What is a .Net Business Component?

A .Net Business Component is an artifact realized on the .Net platform, adhering to the definition of a Business Component described in the reference architecture of the Autonomous Province of Bolzano. It is not *the* Business Component as defined in the reference architecture, but one of its possible technical implementations, realized on the .Net platform, whereby it may implement either the whole Business Component or parts of it only.

According to the architectural guidelines it has the following characteristics:

- It offers services of a well defined business area
- It is hosted by a technical infrastructure which controls access to its services (CEE.Net)
- It is distributable to different machines in order to balance the load

4. Building Plan

The technical implementation of a (part of a) Business Component on the .Net platform at compile time results in a so called *common assembly,* and as many assemblies as tiers are going to be implemented.

At design time, the .Net implementation of a Business Component consists in as many Visual Studio projects as assemblies have to be produced: one for the common assembly, and one for each tier that is going to be implemented.

4.1. The Visual Studio Solution

Every .Net implementation of a Business Component becomes a Visual Studio solution, named after the Business Component itself, and containing at least:

- one Visual Studio project for the types needed by clients in order to contact the Business Component (this is the Business Component's common stuff);
- as many Visual Studio projects as tiers of the Business Component are going to be realized (from one to four);
- a folder named *Components* for third party assemblies referenced by any of the solution's projects.

Note

Note that projects of the solution should reference assemblies only from the following locations:

- .Net framework
- the local CEE.Net's components folder
- the solution's Component folder

The Visual Studio solution may contain additional projects, for example for testing purposes.

4.2. The Common Project

The Common-Project takes the name `<BusinessComponent>.Common` and produces the *common assembly* which will contain the following:

- The .Net interface types for each of the Business Component's tiers that are going to be implemented on .Net or whose services are going to be accessed from a .Net client.
- The data types exchanged with the client by the single methods of the Business Component's services.

Optionally, additional serializable types such as enumerations may be defined in the common assembly.

The common assembly will be given away to service consumers, it's public stuff. It describes the services of the Business Component and the types exchanged in service calls, and it allows to implement a service consumer without the need of a working SOA infrastructure. Obviously, the common assembly is also necessary for implementing the service itself and not only for creating one of its consumers.

Note

Note that the entire common assembly must be automatically "translatable" into artifacts of other platforms such as Java. This guarantees immediate interoperability.

The reason why a Business Component is described by one or more platform specific, compiled artifacts (and not, for example, by a WSDL file), is for the transparent proxy feature of the CEEs: a dynamic proxy for a remote client is only created when the service instance really is remote. This typically happens by reading a WSDL file at design time and by compiling a class built according to what found in it. However, the transparent proxying feature of the CEEs does not deal with WSDL files when the service has been deployed to the same machine from where the service is going to be requested, it rather returns a dynamic instance of the implementing artifact. What it returns is hidden behind the interfaces described in the common packages of a Business Component.

4.2.1. References

The Common Project MUST reference the service broker's common assembly (`<root namespace>.Base.Cee.ServiceBroker.Common.dll`) in order to inherit from interfaces provided by the CEE.Net and for dealing with types offered by the CEE.Net. [T.1]

It SHOULD NOT reference other assemblies except such of the .Net framework or such providing so called *abstract service interfaces*[1]. [O.2]

Note

The common project MUST NOT reference a specific version of any assembly as long as it has not been deployed to the .Net framework's GAC! [T.3]

4.2.2. Namespace Hierarchy

The root namespace of the types defined in the Common Project SHOULD be built as follows: [T.4]

`<root namespace>.<Area>.<Project>.<BusinessComponent>.Common`

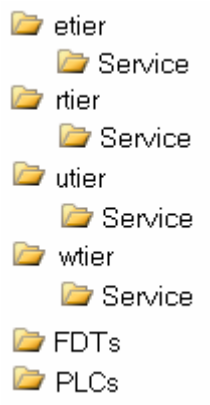
whereby

- `<Area>` defines a wide business area to which the Business Component belongs;

[1] Abstract service interfaces are such that have no real, direct implementation, but other service interfaces inherit from them. What is going to be implemented are the interfaces that inherit from abstract interfaces.

- `<Project>` specifies the project (this is not a Visual Studio project) which has realized the Business Component;
- `<BusinessComponent>` is the name of the Business Component which is (entirely or in part) going to be implemented.

The namespace tree underneath the root namespace MUST contain final leaves called `Service`, SHOULD be structured as follows and MAY be extended: [T.5]



The `<tier>.Service` namespaces will hold the interface types defining the methods of the services offered by the Business Component at the respective tier.

Important

Note that it is crucial that the namespace ends with `.Service`: this is the only semantic rule controlled by the CEE.Net in order to speed up analysis of the assemblies deployed to its components folder.

The `FDTs` namespace will hold the so called *Functional Data Types* (see the reference architecture for a detailed explanation) whose instances are exchanged in service calls that happen at the enterprise tier. Functional Data Types usually are built of class types, but also an enumeration may represent a valid FDT. However, a FDT MUST NOT be anything else than either a class type or an enum type. [T.6]

The `PLCs` namespace will hold the so called *Persistence Language Classes* (see the reference architecture for a detailed explanation) whose instances are exchanged in service calls that happen at the enterprise tier. Persistence Language Classes usually are built of class types, but also an enumeration may represent a valid PLC. However, a PLC MUST NOT be anything else than either a class type or an enum type. [T.7]

4.2.2.1. Service Interfaces

The service interfaces exposed by the Business Component are defined as interface types in the `<tier>.Service` namespace. Every single interface defines a service with multiple methods. Please care about the following rules regarding service interfaces:

Every service interface

- MUST start its name with the uppercase character `I`, and the second character of its name SHOULD be upper case and indicate the tier it belongs to (`R` = resource tier, `E` = enterprise tier, `W` = workspace tier, `U` = user tier). [T.8]

- SHOULD inherit from the CEE.Net's `<root namespace>.Base.Cee.ServiceBroker.ICeeService` interface. [T.9] By inheriting from this interface you provide the CEE.Net at runtime with information for effective service handling (such as association of the service with its Business Component or the tier, and much more important, the fact if a service instance can be reused for multiple requests). If you don't, the CEE.Net will nor care about this information at runtime with a possible loss of performance.

- MAY inherit from another service interface. [T.10] This is useful whenever you either want to extend an existing service interface or if you want your service have to implement a so called abstract service interface for service switching at runtime.

- MAY inherit from the CEE.Net's `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface. [O.11] This qualifies a service for being exposed by the Web Service publisher of the CEE.Net Server Edition to remote clients. Note that service interfaces MUST directly inherit from this interface if you want them have to be published by the CEE.Net. [T.12] In the scenario where interface `A` inherits from interface `B` and indirectly inherits from `IRemoteable` because of the fact that `B` does, a service implementation of `A` will not be published by the CEE.Net to remote clients:

  `A : B` and `B : IRemoteable` ⇨ B is published, A is not
  `A : B, IRemoteable` and `B` does not ⇨ A is published, B is not
  `A : B, IRemoteable` and `B : IRemoteable` ⇨ both A and B are published

  This is in order to allow NOT to publish a more specialized service interface to remote clients although the parent interface is qualified for publishing.

- MUST NOT contain any properties. [T.13]

- MUST NOT use any types as method parameters or method return types other than [T.14]

  - either explicitly defined in its own common assembly (in the `FDTs` or `PLCs` namespace),
  - or such defined in the service broker's common assembly,
  - or types known to be primitive according to the W3C definition

- MAY USE one dimensional arrays of any of the allowed types as input- or output parameters [T.15], but MUST NOT use multidimensional arrays. [T.16]

- SHOULD define an array of the CEE.Net's `<root namespace>.Base.Cee.ServiceBroker.FDT.BusinessResult` type as the return type for each of its methods. [O.17]

- MAY use both `in`- and `out` parameters in its methods [T.18], but MUST order them correctly (in before out). [T.19] Ordering them in a different way will prevent the service from being published by the CEE.Net Server edition even when inheriting from `IRemoteable`.

- MAY use `ref` parameters in its methods. [T.20] When the service interface is remoteable and published to remote clients, such parameters will be `in` parameters. Defining `in`-parameters as ref-parameters is for complying with a COM interoperability issue: for some types (e.g. for arrays in general) the way how COM passes parameters to methods is by default by reference, and COM has no possibility to operate in another way. On the other hand, the "same" types

.Net passes to methods as by value. The .Net COM InteropServices don't care about this and propagate from COM to .Net as COM has received the parameter. So, if COM decides to accept an array only by reference, the InteropServices will do the same when passing the marshaled object to the .Net method. So if you don't define the parameter in your interface as `ref`, the COM InteropServices will fail on searching for a way to pass it by reference since your method expects it as by value (which is .Net's default).

- MAY use nullable primitive types as return- or parameter types by the means of .Net's generics. [T.21]
  This is for complying with a Java interoperability issue: not all primitive types of one platform have primitive equivalents in another platform. Java's Date for example is not a primitive type and therefore nullable, while .Net's `DateTime` equivalent is primitive and cannot assume a null value. What happens is that a service implemented on the Java platform may send a null when a .Net based consumer expects a `DateTime`, and the service call will fail. When describing the service interface via WSDL, no information about nullability is provided. Nevertheless you can declare a primitive type to be nullable in your service interface, since the service broker will resolve the conflict for you at runtime (the dynamic service proxy is built from the WSDL and forced to implement your service interface, and this would fail if the WSDL does not declare a nullable parameter, while your interface does).

- MAY be decorated with the `System.Runtime.InteropServices.Guid("<guid>")` attribute for COM interoperability [T.22], but this SHOULD be done only in the `etier.Service` namespace. [O.23]

- MAY decorate single methods with the `System.Runtime.InteropServices.COMVisible(false)` attribute in order to hide the method from COM. [T.24]
  This is useful when the method is built in a way that is incompatible with COM (such as `in`-arrays not defined as `ref` or types which can't be marshaled automatically).

If a service interface inherits from and extends another one, and if you have to guarantee COM interoperability, according to the COM interoperability guidelines you SHOULD do one of the following: [T.25]

- Either you repeat in your service interface all the methods of the interface from which it inherits

- Or you define an additional interface for COM which contains, in addition to the ones defined by your service, all the methods defined in parent interfaces

4.2.2.2. Functional Data Types (FDTs)

Functional Data Types are classes whose instances are exchanged between the client and any service of the Enterprise tier. They are defined either as class types or as enum types in the `FDTs` namespace. Along with types defined by W3C to be primitive and such provided by the CEE.Net itself, they are the only types allowed to be used by a service of the enterprise tier. FDTs are thought for nothing else than data transport and have the following characteristics:

- They MUST NOT contain anything else then properties. [T.26]

- The types of their properties MUST NOT be anything else than either W3C primitive or another FDT of the own common assembly [T.27] (they MAY be nested [T.28]).

- They MUST provide both a default getter (simply returns a local variable) and a default setter (simply assigns the value to a local variable) for each of its

properties; [T.29]
they MUST NOT implement any logic neither in the getter nor the setter. [T.30]

- MAY use nullable primitive types as properties by the means of .Net's generics. [T.31]
  This is for complying with a Java interoperability issue: not all primitive types of one platform have primitive equivalents in another platform. Java's Date for example is not a primitive type and therefore nullable, while .Net's `DateTime` equivalent is primitive and cannot assume a null value. What happens is that a service implemented on the Java platform may send a null when a .Net based consumer expects a `DateTime`, and the service call will fail. When describing the service interface via WSDL, no information about nullability is provided. Nevertheless you can declare a primitive type to be nullable in your service interface, since the service broker will resolve the conflict for you at runtime (the dynamic service proxy is built from the WSDL and forced to implement your service interface, and this would fail if the WSDL does not declare a nullable parameter, while your interface does).

- They MUST be decorated with the `System.Runtime.InteropServices.Guid("<guid>")` attribute for COM interoperability if used by any method which is visible to COM. [T.32]
  They MAY NOT be decorated with this attribute if they are not used by any method which is visible to COM. [T.33]

- They MAY implement an interface defined either in the own common assembly (in the FDTs namespace) or in the FDTs namespace of the assembly offering a parent interface of the service. [T.34]
  This is useful when dealing with so called abstract service interfaces for service switching at runtime.

- FDTs MAY inherit from each other. [T.35]

4.2.2.3. Persistence Language Classes (PLCs)

Persistence Language Classes are classes whose instances are exchanged between the client and any service of the resource tier. They are defined either as class types or as enum types in the `PLCs` namespace. Along with types defined by W3C to be primitive and such provided by the CEE.Net itself, they are the only types allowed to be used by a service of the resource tier. PLCs are thought for nothing else than data transport, and they have the same characteristics as FDTs.

4.2.2.4. Enumerations

Enumerations are allowed in common assemblies and service interfaces as long as their members compile to the underlying primitive type (typically an integer). They are defined as enum types either in the `FDTs` or in the `PLCs` namespace.

4.2.3. Build Properties

When compiling the release version of the project, the following properties MUST be set [O.36]:

- *Register for COM interop*, if your services have to be COM visible
- *XML documentation file:* check this option and set its value to `bin\Release\<root namespace>.<Area>.<Project>.<BusinessComponent>.Common.XML`

4.2.4. XML Comments

All public classes of the common project and their public members MUST be XML documented according to the following documentation guidelines: [O.37]

- The summary of each member explains in a *concise* way what the member is about. In particular care about the following rules:
  - For properties, specify if they offer a getter or a setter or both of them: if they have only a getter, start the summary with "*Gets …*", if they have only a setter, start the summary with "*Sets …*", and if they have both of them, start the summary with "*Gets or sets …*".
  - For methods, specify what the operation is about: if anything is returned, start the summary with "*Returns …*", else simply describe what the method does.
- Describe each of a method's parameters in a *concise* way.
- Describe in a *concise* way the method result (if any is returned).
- List all the custom exceptions thrown by any method or the getter or the setter of any property, and explain when they can happen.
- Explain all further details regarding the member in the remarks section.

It is important to be concise with summary description and not to use advanced formatting options since the XML documentation is read by Visual Studio's IntelliSense feature at design time, and IntelliSense has some limitations.

4.2.5. Compiled Help Files

The XML documentation of a Business Component's public interfaces – all contained in its common assembly – MUST be compiled [O.38] into a `*.chm` file; to accomplish this, the Sandcastle Help File Builder SHOULD be used with the following customized project properties [O.39]:

- *Assemblies to document*: the Business Component's common assembly along with its XML documentation file;
- *Dependencies*: all assemblies referenced by the Business Component's common assembly;
- *Copyright Text*: `(c) Autonomous Province of Bolzano <year started> - <year ended>`
- *HelpTitle*: `<Name of Business Component> - Interfaces Class Library Documentation`
- *HtmlHelpName*: `<Name of Business Component> - Interfaces Class Library Documentation`
- *PresentationStyle*: `vs2005`
- *SdkLinkType*: `None`

4.2.6. Output

When the common project is compiled, a single assembly with the name `<root namespace>.<Area>.<Project>.<BusinessComponent>.Common.dll` MUST be produced. [T.40]
This file has to be referenced by any project which needs access to the Business Component's services.

4.2.7. Assembly Information

The assembly, produced by compiling the Common project, MUST have set the following properties: [O.41]

- Title: `<BusinessComponent>.Common`
- Description: a short description of what the Business Component is about
- Company: `Autonomous Province of Bolzano`
- Product: `<BusinessComponent>.Common`
- Copyright: `Copyright © Autonomous Province of Bolzano <year>`
- Assembly Version: Please refer to the *Application Versioning.pdf* document
- File Version: Please refer to the *Application Versioning.pdf* document

4.3. Tiers and Projects

The following rules apply to Business Components in general, independently from a specific implementation. The rules saying that something *must* happen mean that *any* implementation *must* respect them, but it may not necessarily be your .Net implementation to realize it. But if it is yours, you have to adhere to the rules.

Every Business Component is made up of at most four tiers. Please note that there's a difference between a tier and a layer: while tiers aim at physical distribution of software artifacts for load balancing, layers represent a logical grouping of Business Components for ruling dependencies between service calls.

The stack of tiers that finally constitutes the technical implementation of a Business Component is compound of the following elements:

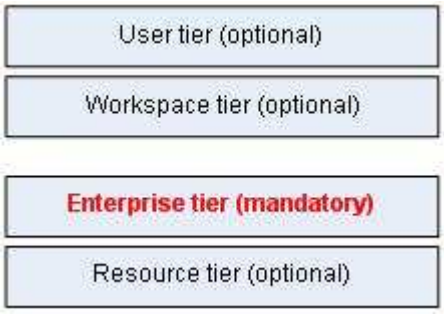


Figure 1: Tiers of Business Component

While layers exist for ruling which Business Component can contact another, tiers imply an analog dependency at a lower level, within a single Business Component:

- Optional tiers (resource tier, workspace tier, user tier) MAY be implemented. [O.100]
- Mandatory tiers MUST be implemented. [O.101]
- Tiers at a higher level MAY call tiers at a lower level, but tiers at a lower level MUST NOT call tiers at a higher level. [O.102]
- When calling tiers of a lower level, implemented tiers MUST NOT be skipped. [O.103]
- The policy about distribution of a Business Component implementation decides about "remoteability" of services:
    - Every tier that is thought to be deployed to a server other than the one that is expected to call it MUST implement the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface. [T.104]
    - Services at the user tier MUST NOT implement the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface. [T.105]
    - Services at the enterprise tier SHOULD implement the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface. [O.106]

Typically, the resource tier is deployed to the same server as the enterprise tier, and has therefore not to be remoteable. The same can be said about the user- and the workspace tiers.

Calls between Business Components MUST NOT be done from any tier to tier other than from enterprise tier to enterprise tier. [O.107]

Since tiers are realized for distributing an implementation of a Business Component to different servers and computers, they must compile to different assemblies. This implies that every tier that is going to be implemented becomes a single project within the Visual Studio solution (which represents the Business Component).

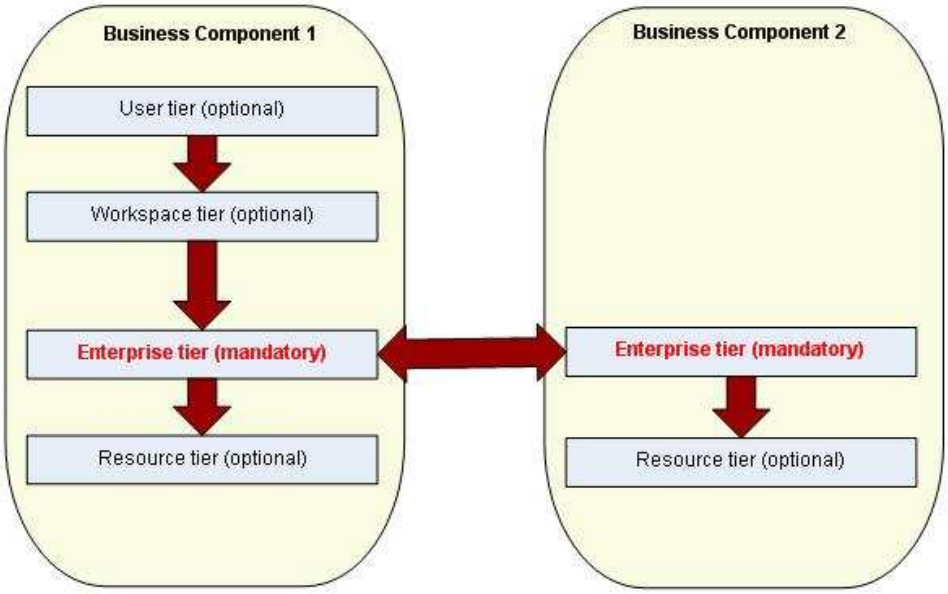


Figure 2: Intra- and inner-communication of Business Components

4.3.1. Dependencies

The single tier implementations generally depend on:

- The Common project which provides the service interfaces
- The CEE.Net which provides base services, the service broker and business results

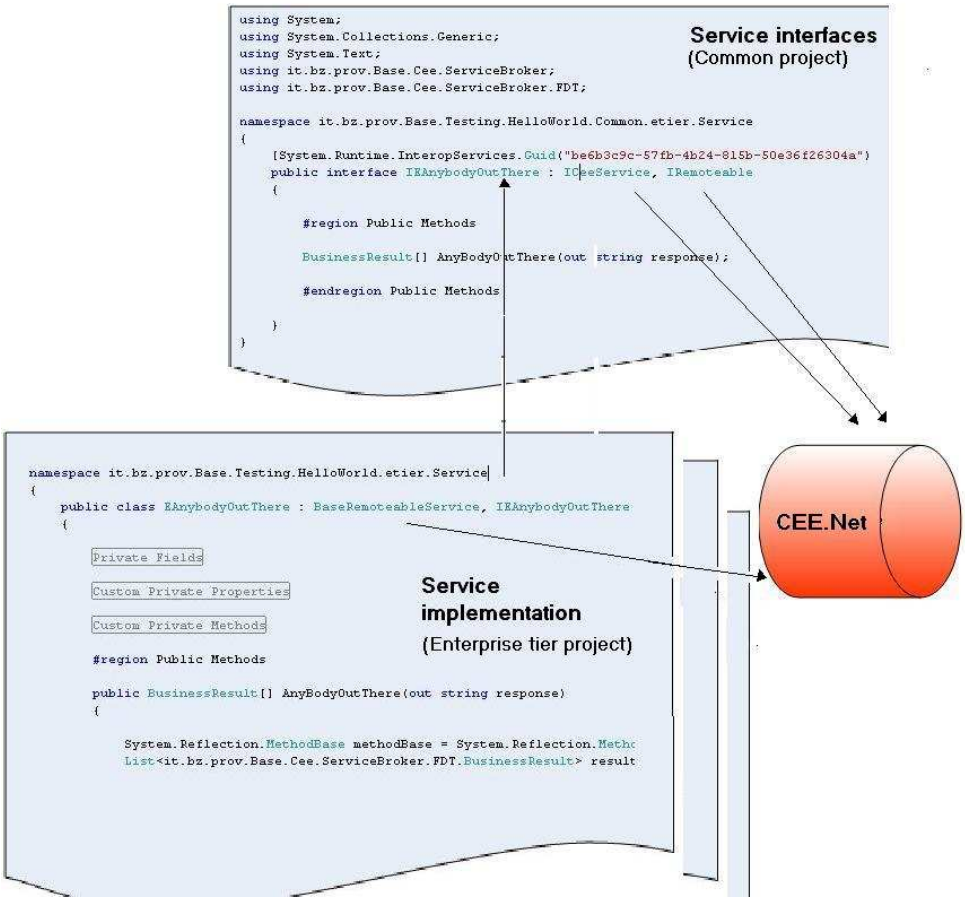


Figure 3: A tier implementation depends on the common assembly and the CEE.Net

Within a Visual Studio project, these dependencies translate into project- or file references.

For deployment scenarios the dependencies mean that both a service provider and a service consumer must have a *local* CEE.Net and the common assembly of the Business Component side by side.

4.3.2. The Resource Tier Project

The resource tier has to deal with persistence logic (and nothing else). It is up to this tier to receive from the enterprise tier the data that has to be saved, and to preprocess it in order to pass it finally to the underlying data store, and vice versa for reading in order to pass data to the enterprise tier. Preprocessing or postprocessing includes all the things that have to be done in order to be efficient with data handling (such as normalization / denormalization if the underlying data store is a relational database, flag translation etc.).

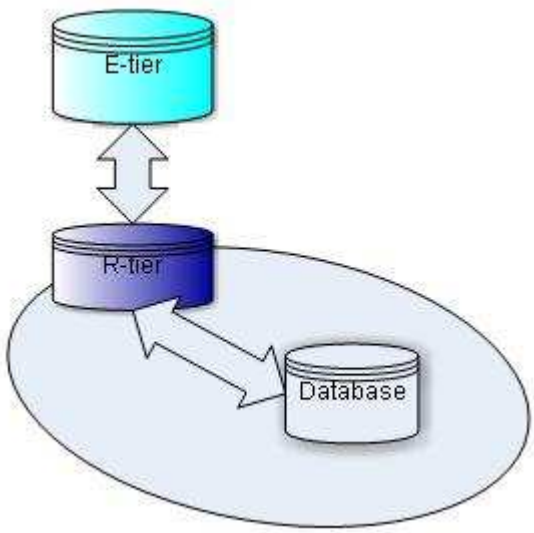


Figure 4: The resource tier connects to an external data store

Note

Note that the resource tier does no business logic!

4.3.2.1. References

The resource tier project

- MUST reference the service broker's common assembly (`<root namespace>.Base.Cee.ServiceBroker.Common.dll`) in order to inherit from base CEE.Net services and for dealing with types offered by the CEE.Net. [T.201]
- MUST reference the Business Component's *Common Project* in order to implement the services of the resource tier. [T.202]
- MUST NOT reference common assemblies of other Business Components except so called abstract ones or such of the Foundation- or the Auxiliary layer. [O.203]
- MAY reference assemblies of the .Net framework or third party assemblies for accomplishing the persistence tasks. [T.204]

Note

The resource tier project MUST NOT reference a specific version of any assembly as long as it has not been deployed to the .Net framework's GAC! [T.205]

4.3.2.2. Namespace Hierarchy

The root namespace of the types defined in the resource tier project SHOULD be built as follows: [T.206]

`<root namespace>.<Area>.<Project>.<BusinessComponent>.rtier`

whereby

- `<Area>` defines a wide business area to which the Business Component belongs;

- <Project> specifies the project (this is not a Visual Studio project) which has realized the Business Component;
- <BusinessComponent> is the name of the Business Component which is (entirely or in part) going to be implemented.

The namespace tree underneath the root namespace SHOULD be structured as follows and MAY be extended: [T.207]

4.3.2.2.1. The Service namespace

The Service namespace will contain the service implementations of the resource tier (this is the implementation of the interface types defined in the common assembly's rtier.Service namespace). To each service implementation the following rules apply:

- it SHOULD inherit from a base service provided by the service broker's common assembly (usually <root namespace>.Base.Cee.ServiceBroker.BaseLocalService; if the resource tier is remoteable <root namespace>.Base.Cee.ServiceBroker.BaseRemoteableService). [T.208]
- it MUST implement exactly one of the service interfaces, defined in the rtier.Service namespace of the Business Component's Common Project. [T.209]
- it MAY implement an explicit COM interface defined in the Business Component's Common Project. [T.210]
- it SHOULD take the same name as the service interface except the leading I (for example, if the service interface has the name IRMyService, the service name becomes RMyService). [T.211]
- it SHOULD be done in a single file named <servicename>.cs. [T.212]
- it MUST provide a default constructor (this is, without constructor parameters), but it MAY provide additional ones. [T.213]
- it SHOULD provide a constructor that takes a serialized CEE.Net type library as a constructor argument in order to be suitable for being used by the CEE.Net itself: public RMyService(object typeLibrary). [T.214]
- each of its methods SHOULD log – apart from its regular actions - its entry along with its input parameters, and its exit with its return value(s). [T.215]
- each of its methods SHOULD control access to protected resources. [O.216]
- it SHOULD allocate its resources (such as data stores) dynamically. [O.217]
- its types and type members SHOULD NOT be COM visible. [T.218]

4.3.2.2.2. The Events namespace

The Events namespace contains types that allow to structure the events that are worthy to be logged. Please refer to the chapter about Implementation Patterns for logging.

4.3.2.2.3. The Results namespace

The `Results` namespace contains types that allow to structure and to handle the business results returned by the single methods of the service to implement. Please refer to the chapter about Implementation Patterns for result handling.

4.3.2.3. Output

When the resource tier project is compiled, a single assembly with the name `<root namespace>.<Area>.<Project>.<BusinessComponent>.rtier.dll` MUST be produced. [T.219]

4.3.2.4. Assembly Information

The assembly, produced by compiling the resource tier project, MUST have set the following properties: [O.220]

- Title: `<BusinessComponent>.rtier`
- Company: `Autonomous Province of Bolzano`
- Product: `<BusinessComponent>.rtier`
- Copyright: `Copyright © Autonomous Province of Bolzano <year>`
- Assembly Version: Please refer to the *Application Versioning.pdf* document
- File Version: Please refer to the *Application Versioning.pdf* document

4.3.3. The Enterprise Tier Project

The enterprise tier implements the business logic of the Business Component without caring about presentation- or persistence tasks. The services of this tier will neither care about the way how the input they need or the output they produce is stored in a container (such as a relational database), nor will they care about how data will be presented to the user (especially, they will not behave in a special way in order to satisfy a presentation tool).

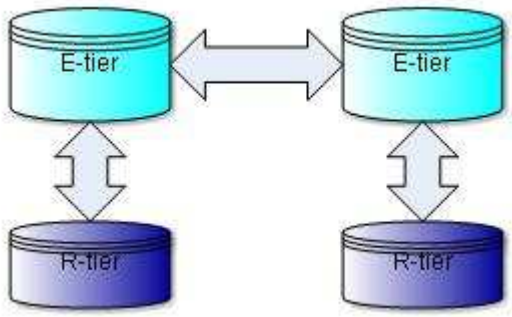


Figure 5: Enterprise tier communication

4.3.3.1. References

The enterprise tier project

- MUST reference the service broker's common assembly (`<root namespace>.Base.Cee.ServiceBroker.Common.dll`) in order to inherit from base CEE.Net services and for dealing with types offered by the CEE.Net. [T.301]

- MUST reference the Business Component's *Common Project* in order to implement the services of the enterprise tier. [T.302]
- MAY reference any other common assembly. [O.303]
- MAY reference assemblies of the .Net framework or third party assemblies for accomplishing the business logic tasks. [T.304]

Note

The enterprise tier project MUST NOT reference a specific version of any assembly as long as it has not been deployed to the .Net framework's GAC! [T.305]

4.3.3.2. Namespace Hierarchy

The root namespace of the types defined in the enterprise tier project SHOULD be built as follows: [T.306]

```
<root namespace>.<Area>.<Project>.<BusinessComponent>.etier
```

whereby

- `<Area>` defines a wide business area to which the Business Component belongs;
- `<Project>` specifies the project (this is not a Visual Studio project) which has realized the Business Component;
- `<BusinessComponent>` is the name of the Business Component which is (entirely or in part) going to be implemented.

The namespace tree underneath the root namespace SHOULD be structured as follows and MAY be extended: [T.307]

4.3.3.2.1. The Service namespace

The `Service` namespace will contain the service implementations of the enterprise tier (this is the implementation of the interface types defined in the common assembly's `etier.Service` namespace). To each service implementation the following rules apply:

- it SHOULD inherit from a base service provided by the service broker's common assembly (usually `<root namespace>.Base.Cee.ServiceBroker.BaseRemoteableService`, if the enterprise tier is not remoteable `<root namespace>.Base.Cee.ServiceBroker.BaseLocalService`). [T.308]
- it MUST implement exactly one service interface defined in the `etier.Service` namespace of the Business Component's Common Project. [T.309]
- it MAY implement an explicit COM interface defined in the Business Component's Common Project. [T.310]
- it SHOULD take the same name as the service interface except the leading `I` (for example, if the service interface has the name `IEMyService`, the service name becomes `EMyService`). [T.311]
- it SHOULD be done in a single file named `<servicename>.cs`. [T.312]

- it MUST provide a default constructor (this is, without constructor parameters), but it MAY provide additional ones. [T.313]
- it SHOULD provide a constructor that takes a serialized CEE.Net type library as a constructor argument in order to be suitable for being used by the CEE.Net itself: `public EMyService(object typeLibrary).` [T.314]
- each of its methods SHOULD log – apart from its regular actions - its entry along with its input parameters, and its exit with its return value(s). [T.315]
- each of its methods SHOULD control access to protected business functionality. [O.316]
- its types and type members MAY be COM visible. [T.317]

4.3.3.2.2. The Events namespace

The `Events` namespace contains types that allow to structure the events that are worthy to be logged. Please refer to the chapter about Implementation Patterns for logging.

4.3.3.2.3. The Results namespace

The `Results` namespace contains types that allow to structure and to handle the business results returned by the single methods of the service to implement. Please refer to the chapter about Implementation Patterns for result handling.

4.3.3.3. Output

When the enterprise tier project is compiled, a single assembly with the name `<root namespace>.<Area>.<Project>.<BusinessComponent>.etier.dll` MUST be produced. [T.318]

4.3.3.4. Assembly Information

The assembly, produced by compiling the enterprise tier project, MUST have set the following properties: [O.319]

- Title: `<BusinessComponent>.etier`
- Company: `Autonomous Province of Bolzano`
- Product: `<BusinessComponent>.etier`
- Copyright: `Copyright © Autonomous Province of Bolzano <year>`
- Assembly Version: Please refer to the *Application Versioning.pdf* document
- File Version: Please refer to the *Application Versioning.pdf* document

4.3.4. The Workspace Tier Project

The workspace tier of a Business Component handles the states of interactions happened between a consumer and the Business Component and adds a kind of session handling to a sessionless SOA. This includes for example persistence of intermediary data (when for example a result can be returned to a Business Component only after having collected data asynchronously from heterogeneous sources), caching tasks, or offline synchronization.

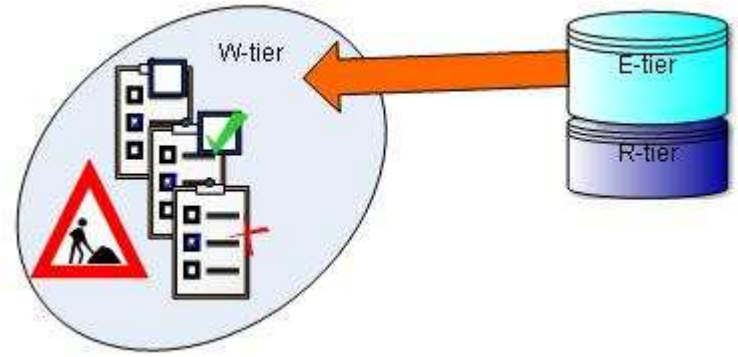


Figure 6: Work in progress at the workspace tier

4.3.4.1. References

The workspace tier project

- MUST reference the service broker's common assembly (`<root namespace>.Base.Cee.ServiceBroker.Common.dll`) in order to inherit from base CEE.Net services and for dealing with types offered by the CEE.Net. [T.401]
- MUST reference the Business Component's *Common Project* in order to implement the services of the workspace tier. [T.402]
- MUST NOT reference other common assemblies except such of the Foundation- or the Auxiliary layer. [O.403]
- MAY reference assemblies of the .Net framework or third party assemblies for accomplishing its tasks. [T.404]

Note

The workspace tier project MUST NOT reference a specific version of any assembly as long as it has not been deployed to the .Net framework's GAC! [T.405]

4.3.4.2. Namespace Hierarchy

The root namespace of the types defined in the enterprise tier project SHOULD be built as follows: [T.406]

`<root namespace>.<Area>.<Project>.<BusinessComponent>.wtier`

whereby

- `<Area>` defines a wide business area to which the Business Component belongs;
- `<Project>` specifies the project (this is not a Visual Studio project) which has realized the Business Component;
- `<BusinessComponent>` is the name of the Business Component which is (entirely or in part) going to be implemented.

The namespace tree underneath the root namespace SHOULD be structured as follows and MAY be extended: [T.407]

4.3.4.2.1. The Service namespace

The `Service` namespace will contain the service implementations of the workspace tier (this is the implementation of the interface types defined in the common assembly's `wtier.Service` namespace). To each service implementation the following rules apply:

- it SHOULD inherit from a base service provided by the service broker's common assembly (usually `<root namespace>.Base.Cee.ServiceBroker.BaseLocalService`). [T.408]
- it MUST implement exactly one of the service interfaces defined in the `wtier.Service` namespace of the Business Component's Common Project. [T.409]
- it MAY implement an explicit COM interface defined in the Business Component's Common Project. [T.410]
- it SHOULD take the same name as the service interface except the leading `I` (for example, if the service interface has the name `IWMyService`, the service name becomes `WMyService`). [T.411]
- it SHOULD be done in a single file named `<servicename>.cs`. [T.412]
- it MUST provide a default constructor (this is, without constructor parameters), but it MAY provide additional ones. [T.413]
- each of its methods SHOULD log – apart from its regular actions - its entry along with its input parameters, and its exit with its return value(s). [T.414]
- each of its methods SHOULD control access to protected business functionality. [O.415]
- its types and type members MAY be COM visible. [T.416]

4.3.4.2.2. The Events namespace

The `Events` namespace contains types that allow to structure the events that are worthy to be logged. Please refer to the chapter about Implementation Patterns for logging.

4.3.4.2.3. The Results namespace

The `Results` namespace contains types that allow to structure and to handle the business results returned by the single methods of the service to implement. Please refer to the chapter about Implementation Patterns for result handling.

4.3.4.3. Output

When the workspace tier project is compiled, a single assembly with the name `<root namespace>.<Area>.<Project>.<BusinessComponent>.wtier.dll` MUST be produced. [T.417]

4.3.4.4. Assembly Information

The assembly, produced by compiling the workspace tier project, MUST have set the following properties: [O.418]

- Title: `<BusinessComponent>.wtier`

- Company: `Autonomous Province of Bolzano`
- Product: `<BusinessComponent>.wtier`
- Copyright: `Copyright © Autonomous Province of Bolzano <year>`
- Assembly Version: Please refer to the *Application Versioning.pdf* document
- File Version: Please refer to the *Application Versioning.pdf* document

4.3.5. The User Tier Project

The user tier interacts with the user by the means of a graphical interface. Note that the user tier is not *the* client: the user tier of a Business Component offers a collection of UI-subparts that cover nothing else than the functionality offered by the Business Component (while the client most probably will deal with multiple Business Components). The subparts have to be requested by a UI-container, most probably along with such of other Business Components, and are then assembled to a comprehensible desktop or dashboard.

Obviously, the user tier will never be technology-agnostic, since its artifacts have to be directly referenced by a platform specific container: The platform on which the container is built will be the one to be used for building the Business Component's user tier. Theoretically the user tier of a Business Component could be implemented multiple times, in order to service multiple UI containers built on different platforms.

What the user tier offers are once again services that can be requested from the CEE.Net, although it is quite obvious that they won't be remoteable. Their interfaces can (and will have to) be built in a more complex way in order to allow the container to coordinate them all.

For a deeper understanding of how to request and coordinate user tier services, please refer to *Communication Patterns for utier Services.ppt*.

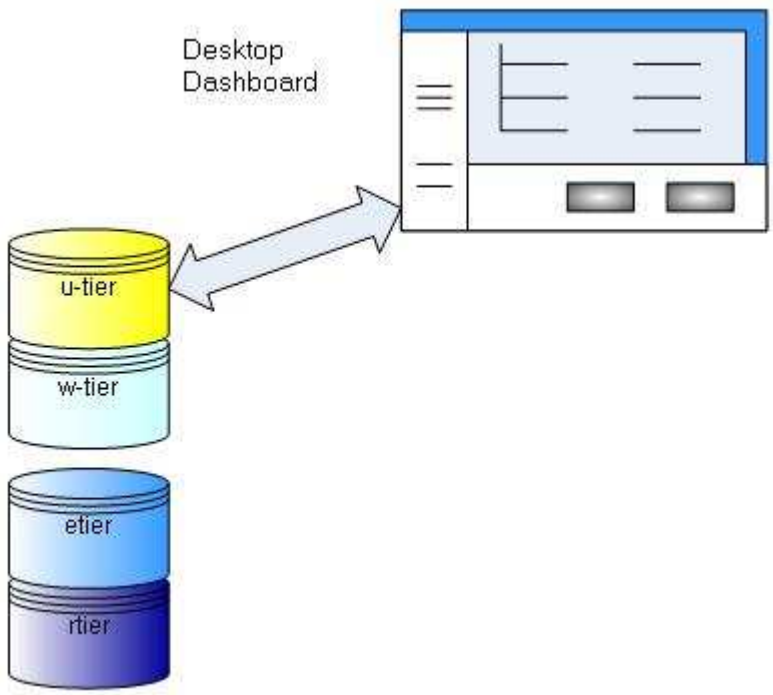


Figure 7: Desktops or dashboards request UI-parts from the Business Component's user tier

4.3.5.1. References

The user tier project

- MUST reference the service broker's common assembly (`<root namespace>.Base.Cee.ServiceBroker.Common.dll`) in order to inherit from base CEE.Net services and for dealing with types offered by the CEE.Net. [T.501]
- MUST reference the Business Component's *Common Project* in order to implement the services of the user tier. [T.502]
- MUST NOT reference other common assemblies except such of the Foundation- or the Auxiliary layer. [O.503]
- MAY reference assemblies of the .Net framework or third party assemblies for accomplishing its tasks. [T.504]

Note

The user tier project MUST NOT reference a specific version of any assembly as long as it has not been deployed to the .Net framework's GAC! [T.505]

4.3.5.2. Namespace Hierarchy

The root namespace of the types defined in the enterprise tier project SHOULD be built as follows: [T.506]

`<root namespace>.<Area>.<Project>.<BusinessComponent>.utier`

whereby

- <Area> defines a wide business area to which the Business Component belongs;
- <Project> specifies the project (this is not a Visual Studio project) which has realized the Business Component;
- <BusinessComponent> is the name of the Business Component which is (entirely or in part) going to be implemented.

The namespace tree underneath the root namespace SHOULD be structured as follows and MAY be extended: [T.507]

4.3.5.2.1. The Service namespace

The Service namespace will contain the service implementations of the user tier services (this is the implementation of the interface types defined in the common assembly's utier.Service namespace). To each service implementation the following rules apply:

- it SHOULD inherit from the appropriate base service provided by the service broker's common assembly (<root namespace>.Base.Cee.ServiceBroker.BaseLocalUserControlService). [T.508]
- it MUST implement exactly one of the service interfaces defined in the utier.Service namespace of the Business Component's Common Project. [T.509]
- it SHOULD take the same name as the service interface except the leading I (for example, if the service interface has the name IUMyService, the service name becomes UMyService). [T.510]
- it SHOULD be done in a single file named <servicename>.cs. [T.511]
- it MUST provide a default constructor (this is, without constructor parameters), but it MAY provide additional ones. [T.512]
- each of its methods SHOULD log – apart from its regular actions - its entry along with its input parameters, and its exit with its return value(s). [T.513]
- each of its methods SHOULD control access to protected business functionality. [O.514]

4.3.5.2.2. The Events namespace

The Events namespace contains types that allow to structure the events that are worthy to be logged. Please refer to the chapter about Implementation Patterns for logging.

4.3.5.2.3. The Results namespace

The Results namespace contains types that allow to structure and to handle the business results returned by the single methods of the service to implement. Please refer to the chapter about Implementation Patterns for result handling.

4.3.5.3. Output

When the user tier project is compiled, a single assembly with the name <root namespace>.<Area>.<Project>.<BusinessComponent>.utier.dll MUST be produced. [T.515]

4.3.5.4. Assembly Information

The assembly, produced by compiling the user tier project, MUST have set the following properties: [O.516]

- Title: `<BusinessComponent>.utier`
- Company: `Autonomous Province of Bolzano`
- Product: `<BusinessComponent>.utier`
- Copyright: `Copyright © Autonomous Province of Bolzano <year>`
- Assembly Version: Please refer to the *Application Versioning.pdf* document
- File Version: Please refer to the *Application Versioning.pdf* document

4.4. Packaging for the CEE.Net

This chapter describes how to package the outputs of the single projects that make up a .Net based implementation of a Business Component, in order to have them deployed on a Windows platform.

.Net uses xcopy-deployment, thus packaging primarily means to establish the correct folder structure for putting then the output assemblies in the right place:

- The common assembly MUST go to the root folder of the package (which will match the components folder of the CEE.Net after the package has been deployed). [T.601] This is important in order to allow services of other locally deployed Business Components to call the ones of the BC that we are going to package.
- The assemblies of the single tiers MAY be put into arbitrary subfolders of the package. [T.602]
- The assemblies of the single tiers MAY be deployed (and therefore packaged) only partially (from 0 to 4 tiers). [T.603]
- If multiple packages are created for a Business Component, all of them MUST contain the common assembly. [T.604]
- If a tier implementation needs a third party assembly that has not been deployed to the GAC, it MUST be packaged side by side the tier implementation (this means, it has to be put into the same folder). [T.605]

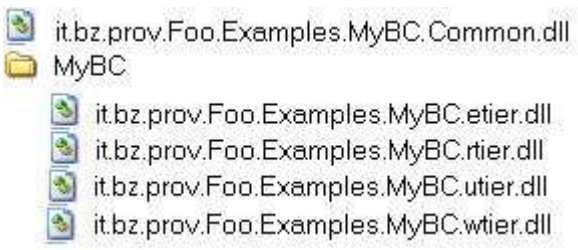


Figure 8: Packaging of a Business Component

Note

There are some aspects to consider when deploying the single tier implementations of a Business Component to subfolders:

Due to the assembly probing algorithm of the .Net framework, assemblies are loaded from side by side the referencing assembly, or from well defined subfolders starting from the referencing assembly's position. But they are never loaded from a folder outside the position of the referencing assembly. This is a problem for tier implementations (which always reference their common assembly) deployed to subfolders and called by another service[2], since the common assembly is found only at a higher level of the folder tree. In this case the referencing assembly raises an exception which is caught by the CEE.Net's service broker. It will then be the service broker to load the requested assembly from the component's folder. This may have negative impact on performance of service requests.

If tier implementations reference third party assemblies the decision about to deploy to subfolders has some implications:

If a third party assembly is used by multiple Business Components which are all deployed to single subfolders, the third party assembly may either be deployed to each of the subfolders or it may be put into the root of the CEE.Net's components folder. The first alternative (deploying multiple times to different subfolders) will increase the workload of the CEE.Net's Kernel which will have to analyze the same assembly multiple times, only to find that they do all not contain any interesting things. On the other hand, this alternative allows to deal with different versions of one single third party assembly: Business Component A could use another version than Business Component B. The second alternative (to put third party assemblies into the root of the CEE.Net's components folder) implies that all Business Components which don't have the third party assembly side by side are forced to use the same version.

As a conclusion we can say that putting tier implementations of Business Components to different subfolders has some (moderate) negative performance impact, but allows to deal with different versions of third party assemblies, whereas putting them into the root of the CEE.Net's components folder will force all Business Components to adhere to a single version of the third party assembly (this may be an advantage).

Be aware of the fact that we are talking only about CEE.Net and Business Components, ignoring the client. Much of the things we said above could be compromised by assemblies loaded by the client which is accessing the Business Component.

4.4.1. Release Package vs. Developer Package

A .Net based implementation of a Business Component SHOULD be packaged in two formats: [T.606]

[2] If the tier implementation is called by a desktop or a dashboard or any other client, the common assembly has already been loaded into the Application Domain by the client itself (which has to reference it in order to have the tier's interfaces); the problem gets evident as soon as service A (whose common assembly has been loaded by the client) calls service B (unknown to the client).

- The release package contains the release builds of the projects described above and will be deployed to the production system of the organization.
- The developer package contains all the stuff contained in the release package, and in addition it contains both compiled and uncompiled XML documentation of the common assembly (which has to be put side by side the common assembly). This package is thought for developers who will access the packaged Business Component and are so provided with IntelliSense for your Business Component and with compiled HTML documentation.

4.4.2. XCopy Deployment

For xcopy deployment it is sufficient to put the Business Component, structured as described above, into a ZIP archive. In this scenario, a package per tier and format SHOULD be created [T.607], each containing the common assembly (and may be its XML documentation) and one single tier implementation.

The packages will then be deployed by the Operations staff (or they will be imported by a developer to his environment).

4.4.3. Windows Installer

An alternative to the xcopy deployment of a Business Component is to package it for the Windows Installer. Such a package SHOULD be structured in features (from 1 to 5), where every feature installs either one of the tiers of the Business Component or only its common assembly. [T.608] The release version and the developer version SHOULD become two different Windows Installer packages. [T.609]

The Windows Installer packages may either be created by your own (or your build environment) by the means of Setup projects (in this case they will have to be validated by the Operations staff in order to guarantee system compliance), or they may be created from your ZIP archives by the Operations staff.

5. Implementation Patterns and Guidelines

The patterns described in the following chapters derive either from technical or from organizational needs. The probably most problematic technical issues will be performance related, and some of the patterns described here help to mitigate the negative impact that the SOA approach in general has on performance. Other implementation patterns take into account organizational needs or try to meet operational requirements.

5.1. Singletons

There are some candidates of objects which pay a lot if handled as singletons, above all the CEE.Net's *service broker*. Every time a service broker is created it requests the type library from the CEE.Net's kernel which is then used to dynamically create instances of local services. If the service broker did not find the requested service among the CEE.Net's local services it will try to find it on a remote machine in order to create a dynamic proxy from the WSDL and couple it with the local common assembly of the Business Component.

All these operations have costs and some negative impact on performance of service calls. Having a single service broker is a great deal, especially for the fact that the broker will cache service instances whenever possible.

You should make sure that every service instance:

- Deals with at most one service broker
- Creates the service broker only when it is needed for the first time

Other candidates are *service instances* which are not shareable (we are talking about services called by your service). When you request a service from the service broker, it reverts to the instance's *shareable* property in order to decide whether to cache the service instance for further requests. So, having a service broker singleton, you automatically have singletons of service instances as long as they are known to be shareable. But it may happen that either the service broker cannot decide whether a service instance is shareable or not (this is, when the service that is going to be called does not implement the `ICeeService` interface) or the service instance is explicitly known not to be shareable (this is generally true for stateful service instances). In this case make sure to maintain the requested service instance all over the lifetime of your service in order to have to request it only once. Note that the minimum cost of a service request is the one of creating the instance dynamically from a local assembly. But neither you nor the service broker will have the control over what happens within the constructor of the service instance: it may do nothing, but just as soon it could try to access a remote resource or do some time expensive stuff.

Within your service implementation, you need a service broker and service instances whenever your service is going to call other services. This is typically the case for all tier implementations, may be except the resource tier (because higher tiers of a Business Component will call lower ones, although a resource tier will most probably need a service broker, too – for accessing services of the auxiliary layer).

Note

Consider inheriting from one of the CEE.Net's base services.

5.2. Logging

Logging is essential for identifying problems at runtime and should be implemented in an exhaustive way.

Attention

Please keep in mind that logging and auditing are not the same! Logging is targeted at identifying technical problems, and therefore tracks the program flow and technical exceptions.

5.2.1. Logging levels (severities)

You should assign different levels to different log entries within your service. Logging levels can be controlled at runtime from the outside and will be used to increase the logging detail if deeper understanding of what is going wrong is needed. The more critical a fact is, the higher your logging level (severity) should be.

You should distinguish in a very coarse way between errors, warnings and information:

- *Errors* should have a quite high severity (you may furthermore distinguish between critical errors and less critical ones) are either technical exceptions or content related (if the result was not the one you did expect, for example for formal or formatting reasons or for missing information). The severity of errors should start from 200;
- *Warnings* should have a medium severity and indicate a fact that is not blocking or falsifying your operation, non the less something to which to pay attention to because not really expected. The severity of warnings should start from 100;
- *Information* should have a low severity and may either log the success of an operation, may be along with some statistical data, or it may track the program flow. Informational logging is often used to understand if something that was expected really did happen, or for performance analysis. The severity of informational log entries should start from 0.

5.2.2. The Logger to use

Although you could use a logging engine of your choice, consider using the logger provided by the base service from which your service implementation inherits. This does not mean that you don't have no longer the choice of your preferred logger, although it will have to be wrapped by a little CEE service: The `Logger` class of any of the CEE.Net's base services attaches to a suitable logging service, and you can specify the one you would like to use. For being suitable, a logging service must be deployed to the local CEE.Net, and it must implement the `<root namespace>.Base.Cee.ServiceBroker.Logging.ILogger` interface. By default, any base service of the CEE.Net uses the `<root namespace>.Base.InformationServices.Logging.IELogManager` service which is included in any of the CEE.Net editions.

The `ILogger` interface of the CEE.Net may hide a lot of sophisticated features of an advanced logging service, on the other hand it makes logging easier and, above all, it allows to control the logging level of your service instance from the outside through the service broker. This is why we recommend to use the `ILogger` of the CEE.Net.

5.2.3. Log Events

Log Events is essentially the stuff that is going to be logged. Especially for the fact that one single event could be found multiple times when your service executes, we recommend not to use simple strings as a log description but rather to structure them in a specific way.

5.2.3.1. Logging language

The language of the single log entries should be American English.

5.2.3.2. Method Entry / Method Exit

In order to track at least when your service enters or exits a method (in order to understand the program flow), the first action your methods should do is to log their entry, and the last before returning the return value should be to log the exit. Please note that it is not the calling method to log that it is going to call any other method, but it is the method that has been called that logs this fact.

When logging the method entry consider logging of the parameters and their values that have been passed to the method. This could be done in this way, with a suitable method `LogEnteringMethod(MethodBae methodBase, object[] parameters)` for logging of the parameter values:

```
Using System.Reflection.;

private String MyMethod(String param1, Boolean param1)

{

    MethodBase methodBase = MethodBase.GetCurrentMethod();

    LogEnteringMethod(methodBase, new object[] { fileName, allUsers });

    …

}
```

When logging the method exit consider logging the return value.

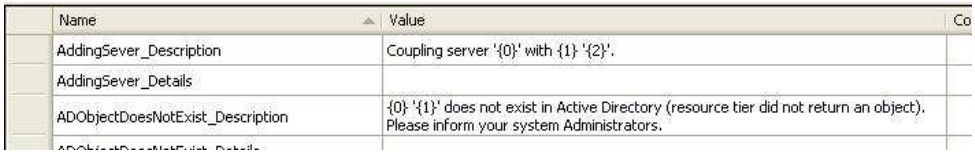
5.2.3.3. Custom Events

A log entry should at least consist of an *identifier*, a *log message* and eventually of explaining *details*. Due to the fact that you could have to log one single event multiple times, we recommend to structure the log events in the following way:

- Create an enum that lists all your events that you expect to occur; the identifier of the log event will match the base type element of your enum item (it is recommended to start with custom log events from ID 1000):

```
internal enum LogEventID : int
{
    InitializingWithPrestagedValues = 0,
    BindInformation = 1,
    EnteringMethod = 2,
    ExitingMethod = 3,
```

- Put the strings to log in a resource file (the strings may contain placeholders for string formatting). Identify the string within the resource file by its enum-item expanded by something that identifies the type of the string to request (such as *Description* or *Details*). Whether to put the severity or not in the resource file depends on the fact if all of your log events that you expect to happen are always of the same severity. If you know some events to be more critical in one context that in another, you should assign the severity by the logging method:



Name	Value	Co
AddingSever_Description	Coupling server '{0}' with {1} '{2}'.	
AddingSever_Details		
ADObjectDoesNotExist_Description	{0} '{1}' does not exist in Active Directory (resource tier did not return an object). Please inform your system Administrators.	

- Create a private method in order to request the logging information - by the means of the enum entry - from the logging resource file, and to pass it to the Logger:

```
private void LogThis(LogEventID logEventId, SeverityTypes severity, string[] arguments)
{
    Data logData = new Data();

    String description = LogEvents.ResourceManager.GetString(logEventId.ToString() + "_Description");
    if (!String.IsNullOrEmpty(description))
    {
        logData.Description = String.Format(description, arguments);

        String details = LogEvents.ResourceManager.GetString(logEventId.ToString() + "_Details");
        if (!String.IsNullOrEmpty(details))
            logData.Details = String.Format(details, arguments);

        logData.Id = (int)logEventId;
        logData.Severity = (int)severity;
        Log(logData);
    }
}
```

Note

Consider inheriting from one of the CEE.Net's base services.

5.3. Service Constructors

Every service implementation must provide a default constructor (this is a constructor that does not take any parameters) in order to give the service broker the possibility to create an instance. You should avoid resource intensive operations within the default constructor in order not to retard the service request.

5.4. Exception handling

All exceptions should be handled within the service that you are going to implement. Consider logging and the use of business results in order to tell the caller that something went wrong.

5.5. ICeeService Members

The members of the `<root namespace>.Base.Cee.ServiceBroker.ICeeService` interface are needed by the service broker in order to handle the service instances in an efficient way (although the most of them are only informational). In addition, they tell the publisher of the CEE.Net server edition how to publish the service if it is found to be remoteable.

If you are inheriting from a CEE.Net's base service, all you need to do is to initialize its private properties (you do this best in the constructor of the service):

- `_serviceAuthor` specifies what will be returned by the `GetServiceAuthor()` method and represents the organization offering the service;
- `_serviceComponent` specifies what will be returned by the `GetServiceComponent()` method and stores the name of the Business Component offering the service;
- `_serviceDescription` specifies what will be returned by the `GetServiceDescription()` method and provides a comprehensible description of what the service is about;
- `_serviceInformation` specifies what will be returned by the `GetServiceInformation()` method; it is of the type int and stores one or more of the following flags:
    - *Remoteable* ($2^0 = 1$) - this property is obsolete and will be ignored by the service broker
    - *Shareable* ($2^1 = 2$) – this property tells the service broker if to cache the service instance if intantiated locally
    - *Stateful* ($2^2 = 4$) – this property tells the service publisher if the service instance should be published in a stateful way (it is ignored by the service broker)
    - *SupportsAsynchronousRemoteCalls* ($2^3 = 8$) – tells the service broker if asynchronous service calls should be supported for the dynamic proxy if the service is called on a remote machine (feature not implemented yet)
- `_serviceLayer` specifies what will be returned by the `GetServiceLayer()` method and tells us which layer the Business Component belongs to (please refer to the Reference Architecture for information about layering);
- `_serviceName` specifies what will be returned by the `GetServiceName()` method and simply denotes the name of the service;
- `_serviceTier` specifies what will be returned by the `GetServiceTier()` method and evidences the tier of the Business Component offering the service;

```
_serviceAuthor = "Autonomous Province of Bolzano";
_serviceComponent = "Popcorn_Services";
_serviceDescription = @"A service that provides functionality related to teachers,
_serviceInformation = 2;
_serviceLayer = "Popcorn_Services";
_serviceName = "RTeachers";
_serviceTier = it.bz.prov.Base.Cee.ServiceBroker.FDT.Tiers.Resource.ToString();
```

If you don't inherit from a CEE.Net's base service you will have to implement the members explicitly.

Note

Consider inheriting from one of the CEE.Net's base services.

5.6. IRemoteable Members

First of all, your service interface has to directly inherit from the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface if you want your service to be remoteable, and on the top of that the members of the interface are needed to accomplish the following tasks:

- Propagate additional and configurable client data by the means of the proprietary SOAP header, starting from the client (this is done by the transparent proxy created by the service broker);
- Propagate client data (both essential information about who the user is and where he is working, and additional information that can be configured at client side and at transport level) between service hops (this is handled by the Web Service wrapper published by the WS Publisher of the CEE.Net's server edition);
- Retrieving client data at server side from the service instance (this is done by the service implementation in order to control access to its functionality or to understand where to search for the resources the caller is interested in).

If you inherit from the `<root namespace>.Base.Cee.ServiceBroker.BaseRemoteableService` service, you don't need to do anything – all the members of the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface are handled by the base service. But if you don't, you will have to implement the members explicitly:

- `GetAuthenticatedComputername()` returns the name of the computer from where the direct service call has started. This is either the client from where the very first request has been sent if you are the first hop, or the last hop which has propagated the request to your service. In any case, the name of the computer returned by this method corresponds to the one used by the user who has *authenticated* against your service, because there are no hops in between of your service and this computer. For implementing the method all you have to do is to return the value of a private variable:

```
public string GetAuthenticatedComputername()
{
    return _authenticatedComputername;
}
```

- `SetAuthenticatedComputername()` sets the name of the computer from where the direct service call has started (see `GetAuthenticatedComputername`). This method is used by the Web Service wrapper (published by the WS publisher of the CEE.Net's server edition) in order to inject its internal service instance, providing it in this way with the information from where the direct service call has

started (this gets the `GetAuthenticatedComputername` method working). For implementing the method all you have to do is to set the value of a private variable:

```
public void SetAuthenticatedComputername(string authenticatedComputername)
{
    _authenticatedComputername = authenticatedComputername;
}
```

- `GetAuthenticatedUsername()` returns the name of the user who has started the direct service call. This is either the user sending the very first request if you are the first hop, or the service user running the Web Service of the last hop which has propagated the request to your service. In any case, the name of the user returned by this method corresponds to the one of the user who has *authenticated* against your service, because there are no hops in between of your service and this user. For implementing the method all you have to do is to return the value of a private variable:

```
public string GetAuthenticatedUsername()
{
    return _authenticatedUsername;
}
```

- `SetAuthenticatedUsername()` sets the name of user who has started the direct service call (see `GetAuthenticatedUsername`). This method is used by the Web Service wrapper (published by the WS publisher of the CEE.Net's server edition) in order to inject its internal service instance, providing it in this way with the information about who started the direct service call (this gets the `GetAuthenticatedUsername` method working). For implementing the method all you have to do is to set the value of a private variable:

```
public void SetAuthenticatedUsername(string authenticatedUsername)
{
    _authenticatedUsername = authenticatedUsername;
}
```

- `GetClientComputername()` returns the name of the computer from where the original service call has started. This is always the client from where the very first request has been sent, regardless of how many hops where between your service and the client. For implementing the method all you have to do is to return the value of a private variable:

```
public string GetClientComputername()
{
    return _clientComputername;
}
```

- `SetClientComputername()` sets the name of the computer from where the original service call has started (see `GetClientComputername`). This method is used by the Web Service wrapper (published by the WS publisher of the CEE.Net's server edition) in order to inject its internal service instance, providing it in this way with the information from where the original service call has started (this gets the `GetClientComputername` method working). For implementing the method all you have to do is to set the value of a private variable:

```
public void SetClientComputername(string clientComputername)
{
    _clientComputername = clientComputername;
}
```

- `GetClientUsername()` returns the name of the user who has started the original service call. This is always the user who sent the very first request, regardless of how many hops where between your service and the client. For implementing the method all you have to do is to return the value of a private variable:

```
public string GetClientUsername()
{
    return _clientUsername;
}
```

- `SetClientUsername()` sets the name of the user who started the original service call (see `GetClientUsername`). This method is used by the Web Service wrapper (published by the WS publisher of the CEE.Net's server edition) in order to inject its internal service instance, providing it in this way with the information who started the original service call (this gets the `GetClientUsername` method working). For implementing the method all you have to do is to set the value of a private variable:

```
public void SetClientUsername(string clientUsername)
{
    _clientUsername = clientUsername;
}
```

Note

Keep in mind that calling within your service implementation any of the `Set<method>` methods described so far won't have any effect except the one that the corresponding `Get<method>` methods of your service instance will return what you passed as a parameter. Doing this is obviously quite silly since you won't do anything else than compromise what your web service wrapper did provide you with (if your service instance is running within a Web Service wrapper). What you set by the means of any of these `Set<methods>` will never be propagated to remote service hops. So there's no need to use them within your implementation (and neither the ones of any service that you are consuming), but you have to implement them in order to allow the service broker and the Web Service wrapper to do their job.

- `GetHeaderProperty(String propertyName)` returns the value of a specific additional property regarding the client, if present. Such properties are propagated in the same way as the ones described above, but except for the decision whether to carry them or not at all they are not controlled by the dynamic proxy created by the service broker. So, the property name is arbitrary and represents a convention established between the client and the service, most probably at a organizational level. Typically these properties are rather infrastructural and of general importance for the services in toto to behave correctly. An example could be the office which a user is currently working for when this makes a difference to the service (may be the service has to access different repositories for different offices).
  When implementing the method you have to deal with pairs of strings, representing the property name and its value respectively. This could be done in

this way (storing the pairs as concatenated strings in a one dimensional String array):

```
public String GetHeaderProperty(String propertyName)
{

    String result = String.Empty;

    foreach (String aProperty in _headerProperties)
    {
        if (aProperty.Contains(";"))
        {
            String name = aProperty.Split(new String[] { ";" }, StringSplitOptions.None)[0];
            String propertyValue = aProperty.Split(new String[] { ";" }, StringSplitOptions.None)[1];
            if (name == propertyName)
            {
                result = propertyValue;
                break;
            }
        }
    }

    return result;

}
```

- `SetHeaderProperties(String[] properties)` stores the additional client properties which the client would like to propagate to the service. You must take care that the `GetHeaderproperties(String propertyName)` method understands what you set. If your `GetHeaderProperty` method is implemented as in the example, all you have to do is to request the client to pass an array of strings to your method where the single entries represent the pairs of property and value:

```
public void SetHeaderProperties(String[] properties)
{
    _headerProperties = properties;
}
```

Note

Consider inheriting from the CEE.Net's `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` service.

5.7. Base Services

It is a good deal to let inherit your service implementation from one of the CEE.Net's base services. They will care about a lot of the recommendations and rules found in this document, especially:

- Inheritance from CEE.Net's interfaces and handling of their members
- Handling of the service broker's singleton instance
- Handling of the logging service

Three base services exist from which your service may inherit:

- `<root namespace>.Base.Cee.ServiceBroker.BaseLocalService` is the right choice when you are implementing a service known not to be remoteable. The base service offers you a service broker singleton, provides you with a Logger,

implements the `<root namespace>.Base.Cee.ServiceBroker.ICeeService` interface in order to guarantee efficient handling of the service's instances by the service broker, and initializes the service properties with default values;

- `<root namespace>.Base.Cee.ServiceBroker.BaseRemoteableService` should be used when you are implementing a service known to be remoteable. It does all what the `BaseLocalService` does, and additionally it implements the `<root namespace>.Base.Cee.ServiceBroker.IRemoteable` interface and handles the latter's methods.
- `<root namespace>.Base.Cee.ServiceBroker.BaseLocalUserControlService` should be your base class if you are implementing a service at the user tier. It does all what the `BaseLocalService` does, and it inherits from the .Net framework's `UserControl` class, suitable for being bound to an arbitrary Windows form.

5.8. Business Result Handling

All methods of your service implementation should return an array of `<root namespace>.Base.Cee.ServiceBroker.FDT.BusinessResult` objects in order to tell the client if the operation went well or wrong (and the client should rely on this answer). Business results are more powerful than SOAP exceptions and describe all the results that may be expected, including non technical errors.

Note

Note that business results do not contain the return or output value of the invoked method. They rather tell you if the method invocation was successful or not, or how reliable or of which quality the return or output values are.

The `BusinessResult` object has the following properties:

- `Id` returns an identifier (an integer) suitable for coupling the result with an event known to the service implementation. The documentation of the service interface should say something about which Id is returned under which circumstances, in order to enable the client to react appropriately.
- `Priority` differentiates between the importance of the single results for the case that more than one result is returned to the client. This makes it possible to the client to deal with more important stuff first.
- `Severity` specifies how critical a result is. Typically a direct connection exists between severity and priority, but priority may put results of the same severity in a prioritizing order.
- `Descriptions` returns an array of localized `ResultDescription` objects as a tribute to multilingual environments. Each entry of the array describes in a different language what has happened. The `IsoTwoLetterCulture` property of the `ResultDescription` object tells which language we are dealing with.
- `AdditionalInfo` exposes an array of strings which can be used by the service to provide the client with additional information about the result, such as the identifiers of the objects that did cause any problems at server side. The format

of the property's strings must be described in detail by the service interface documentation.

As the `BusinessResult` class is well defined, GUI frameworks could provide you with a default result viewer.

It is recommended to handle the Ids of `BusinessResult` objects using an enumeration (the Ids will correspond to the base type element of the enum items), and to manage the multilingual strings with resource files.

5.9. Resource Allocation

Most Business Components will have a resource tier concerning itself with persisting or recovering data processed at the enterprise tier. A maxim to follow when implementing the resource tier is to completely operate dynamically when connecting to underlying data stores:

- If the underlying resources (such as databases or directories or whatever) are going to be moved by the operations staff to different locations (for example to another server), they should be found automatically without the need of intervening in configuration files;
- If the underlying resources are replicated, the nearest instance has to be found automatically, without the need of intervening in configuration files;
- If the underlying resources are distributed (for example adhering to organizational units), the right instance has to be found, taking in consideration the user's identity.

Obviously, some configuration parameters will however be needed by the resource tier: the name of the database (or a pattern for building it at runtime), the type of the directory, credentials for service logins and so on. But what is key to success is decoupling from physical deployment: the operations staff will hardly rename a database or change the login, but they may have the need to move systems to other servers.

5.9.1. Finding the Nearest Resource

For finding the nearest of replicated resources (nearest to the resource tier obviously) you may rely on technologies provided by the resource itself (such as serverless binding to LDAP directories via DNS srv records, or Distributed File Systems), or you may contact another service for asking for the nearest server offering what you need. Note that we are talking about replicas: all instances are equally valid and suitable to the resource tier service.

For the case that the API provided by the resource itself is does not already do the job for you, it is recommended to fall back on a infrastructural service of your SOA which implements the interface `<root namespace>.Base.Abstract.UserEnvironment.IEUserEnvironment`. Note that this is a so called *abstract* service interface, possibly implemented multiple times. This allows you to hold the implementing service interface as a parameter and to switch between different implementations when deploying to different environments. When requesting for example the implementation provided by the CEE.Net's LocalConfig service, you would do it in this way:

```
String myServiceImplementation =
    "it.bz.prov.Base.ConfigurationServices.LocalConfig.Common.etier.Service.IELocalConfig";

IEUserEnvironment configSrv =
    (IEUserEnvironment)broker.GetService(myServiceImplementation,
    CommonSourceTypes.LocalOnly);
```

Once received the service instance, you can request the resource (most probably a server) by the appropriate method. When searching for servers, you can assume that they have been tagged by the system administrators with roles qualifying them for different areas of responsibility, and one role will be for your resource tier (you should read the roles you need from a configuration source, accessible to the system administrators). So, let's say you need a server playing the role accounting, you could request it this way:

```
String[] myServers = configSrv.GetServers("accounting");
```

It is up o the configuration service that you are using to find the nearest instance if multiple servers were suitable for your resource tier.

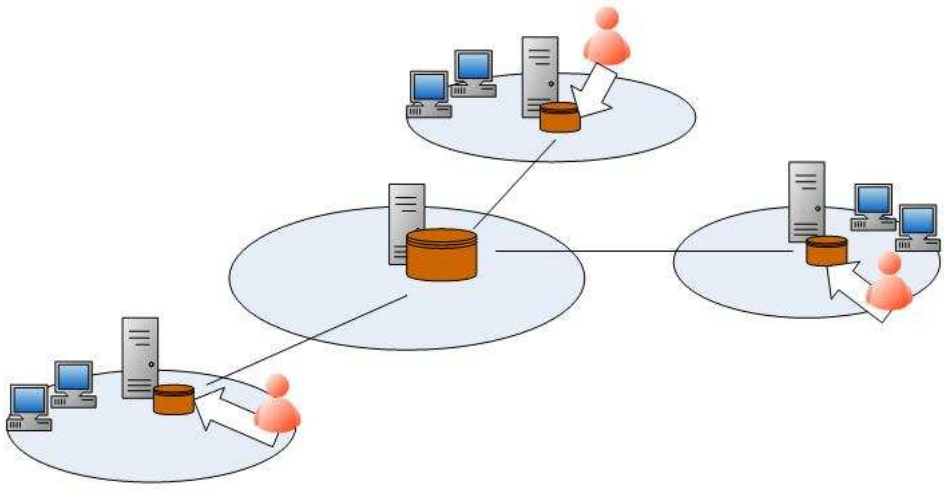


Figure 9: The resource tier always connects to the nearest replica instance of a resource

5.9.2. Finding the Right Instance

Finding the right instance of a distributed resource may be different from finding the nearest one, although ideally the right one should always be near. While "nearest" applies to replicated data stores, "right" applies to distributed ones. Typically, distributed data stores are found together with redundantly deployed service instances (it obviously makes sense having one's resource near the resource tier of a service). It depends on the organizational unit a user belongs to whether a resource is going to be identified to be the right one or not: the data store must hold the data of the user's area of responsibility.

Merely infrastructural services as provided by replicated or distributed resources (such as LDAP directories or Distributed File Systems) don't know anything about organizational areas of responsibility. For finding the right instance once again you will have to fall back on an organizational service that provides you with the resources you are looking for. Any service implementing the abstract service interface `<root namespace>.Base.Abstract.UserEnvironment.IEUserEnvironment` is suitable, and once created an instance (see 5.9.1) you can request a suitable server by calling the appropriate method, specifying the organizational unit you are interested in:

```
String[] myServers = configSrv.GetServers("accounting", "myOffice");
```

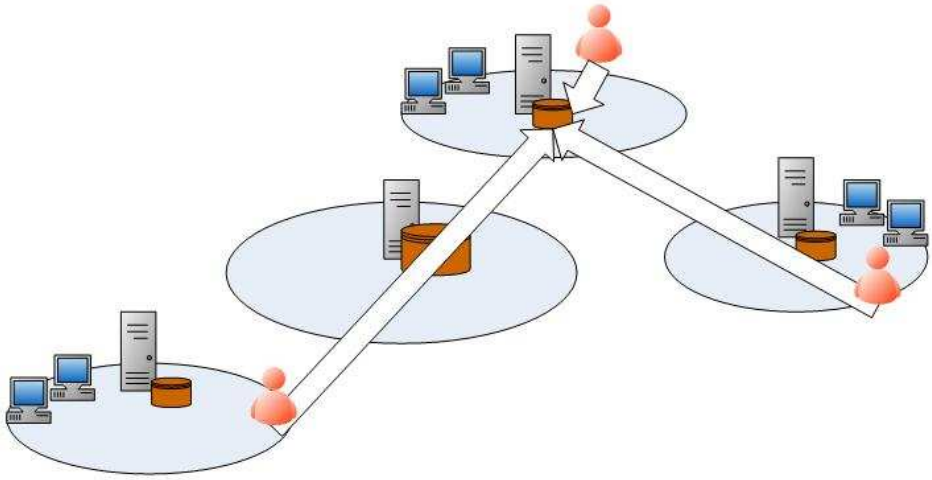


Figure 10: The resource tier always connects to the instance of the user's area of responsibility

5.9.3. Authenticating against a Resource

Whenever a resource that your resource tier is going to access is not natively integrated with your SSO security context (accepting for example Kerberos tokens issued by your KDC), it is recommended to authenticate against the resource with a dedicated service user who has sufficient privileges in order to carry out all of the services' tasks. Access control to the services methods has to be handled by the resource tier implementation and should never be delegated to the resource itself.

5.9.4. Parameterization

Once received the name of the server offering your resource, you will need additional data to access the data store. This may be a connection string along with parameters such as page size, connection timeouts or others, or credentials such as a username or password. Most probably the values of these parameters will be expandable by substituting well defined variables with the names of the resources found dynamically.

You should provide different sections within your configuration files which repeat all or part of the parameters for different environments. This is essential for moving your Business Component from one environment to another because different environments will for example hardly have the same credentials for authenticating against data sources. Your service implementation should dynamically pick the right parameter instance by identifying the domain in which it is currently running. Moving your Business Component from one environment to another is likely to happen: running the Business Component in a testing environment is essential.

Very likely some of these parameters will be have to be protected. Typical candidates are for example credentials of service users. Please refer to 5.11 for how to cipher such parameters.

5.10. Access Control and Auditing

Whenever your service deals with data or functionality that should not be accessible unreservedly, you will have to implement access control in order to protect your service appropriately.

Access control can rely on the following information:

- The user's identity
- The organizational units a user belongs to
- The roles the user plays for a given organizational unit
- The place from where the service call starts

Note

Please keep in mind that access control is different from authentication! The CEE infrastructure will however guarantee that only authenticated users have access to your services. But talking to an authenticated user does only mean talking to somebody you *know*. But the fact that the user is known to the service does not automatically mean that he is allowed to use its functionality.

5.10.1. Getting the User's Identity and Position

For remoteable services, the information who the user is can be requested by calling the `GetClientUsername()` method. In an analogous way the service can understand from where the service call has started by calling the `GetClientComputername()` method. For services which are not remoteable, retrieve the user's identity and the name of the computer from the Operating System.

5.10.2. Analyzing the User's Environment

If the user's identity and his client name are not enough for access control you should request additional information from a service implementing the `<root namespace>.Base.Abstract.UserEnvironment.IEUserEnvironment` interface. Note that this is a so called *abstract* service interface from which other service interfaces inherit with a concrete implementation. When requesting for example the implementation provided by the CEE.Net's *LocalConfig* service, you would do it in this way:

```
String myServiceImplementation =
    "it.bz.prov.Base.ConfigurationServices.LocalConfig.Common.etier.Service.IELocalConfig";

IEUserEnvironment configSrv =
    (IEUserEnvironment)broker.GetService(myServiceImplementation,
    CommonSourceTypes.LocalOnly);
```

The user's environment is mainly characterized by the organizational units he belongs to and by the roles he plays for the single offices. In order to access the user's environment you have to initialize the `IEUserEnvironment` object with the name of the user, and if needed, with the computer which he is working with:

```
configSrv.Initialize("theUsername");
```

or

```
configSrv.Initialize("theUsername", "theComputername");
```

Note

Keep in mind that initializing the `IEUserEnvironment` object with both the user name and the computer name will be slightly slower than initializing it with the username only. So weigh up the disadvantages and advantages

5.10.2.1. When is it important to know where a user is working?

The position of a user who is calling your service may be important in two cases:

- Your service has to understand which of the user's offices are in his range
- Your service has to access a resource near the *user* and not near the position where the service is running

Actually, you can do without in most cases, especially if your service is remoteable: the office which the user is working for can (an should) be propagated by the CEE infrastructure, and you can rely on it without the need to verify how plausible it is what you've received. And a service should generally access resources near its running instance rather than such near the user.

So in most cases you will have to initialize the user's environment with his username only.

5.10.3. Getting the User's Offices

If your service is remoteable and if the CEE infrastructure has been told to propagate the organizational unit which the calling user is currently working for, you can request the user's office by calling the `GetHeaderProperty("office")` method of your service[3].

If your service is not remoteable you will have to provide for the possibility to pass the office as a parameter to the single methods of your service.

5.10.4. Getting the User's Roles

Once created a `IEUserEnvironment` object you can request the list of roles that the user plays for a given organizational unit:

```
IRoleFDT[] roles = configSrv.GetRoles("myOffice");
```

If you only need to know if a specific role is played by the user within a given organizational unit you can check this in this way:

```
Boolean hasRole = configSrv.HasRole("myRole", "myOffice");
```

5.10.5. Where to Control Access within a Business Component?

Not at all tiers of a Business Component access has to be controlled in the same rigorous way.

[3] The string „office" denotes the configurable name of the header property to return. In your environment the name could differ from the one specified in this document.

5.10.5.1. User- and Workspace Tier

There's no need to control access at the user tier or the workspace tier for the following reasons:

- The user tier cannot offer any business functionality without going through the enterprise tier, and cannot retrieve any data without going through the resource tier; so at the user tier there's nothing to protect.
- If the workspace tier deals with data caching in offline scenarios it has to encrypt the cache with the user's identity before persisting it; the same identity will be needed to decrypt it when restoring the session.

5.10.5.2. Enterprise Tier

It is however essential to control access to functionality at the enterprise tier since applications should never contact services of other tiers.

5.10.5.3. Resource Tier

Whether to control access also at the resource tier depends on who is calling it and if it is remoteable or not:

- If the resource tier is remoteable, you can control access basing on the user's identity, his position and his office that it has been provided with – just as any other remoteable service would do;
- If the resource tier is not remoteable you should try to understand if the calling process runs with the identity of a service user or not:
  - If the resource tier understands that it is being called by a service user we can be sure that the original caller has been a remote client (since a client process will never run with a service user), and as your resource tier is not remoteable, the call must have gone through your enterprise tier (because only processes running on your machine were able to call your service); in this case simply trust the enterprise tier – there's no need to redo the access control checks. In order to understand if your process is running in a hop you need a service broker, most probably you already have one for logging or other tasks. Query the service broker's `Hop` property for the information you need;
  - If the resource tier understands that it is not running in a hop, simply retrieve the user name and the computer name from the operating system:

```
if (broker.Hop == HopStates.Yes)
{
    // don't control access
}
else
{
    // control basing on the user's identity
    IEUserEnvironment configSrv =
    (IEUserEnvironment)broker.GetService(myServiceImplementation,
    CommonSourceTypes.LocalOnly);

    configSrv.Initialize(Environment.UserName, Environment.MachineName);
}
```

Note that in this case the `IEUserEnvironment` service that you are using may return multiple available offices for the user, and you will not have the possibility to ask the user for whom he is currently working. In

this case you have no other choice than relying on the office that the user has chosen to work with when he has logged on to the system:

```
String myOffice = configSrv.CurrentUser.GetOffice(OfficeTypes.ChosenAtSystemLevel).Name;
```

5.10.6. Parameterization

It is strongly recommended to store in a configuration file the identities or roles on which your service relies to control access to its functionality or data. This is crucial for decoupling development from operations: system administrators may be required to introduce new roles or to remap roles and permissions due to organizational needs.

Obviously, if you are offered by your organization a more sophisticated system which describes who should be permitted to carry out which tasks (such as a Business Component for Access Control) you are free to use it. Just keep in mind that at the end of the day the operations staff must have the possibility to reconfigure the access policies from the outside.

5.10.7. Auditing

Currently no auditing policy is defined.

5.11. Cryptography

It is strongly recommended to encrypt some security relevant stuff:

- Configuration files of the single tiers of your Business Component: This is most likely to happen for the resource tier which will need access to its repository, but it may occur for every tier that implements any type of access control.
- Cached data at the workspace tier: whenever the workspace tier freezes a user session, the session data should be persisted in a secure an therefore encrypted way. Consider factoring in the user's identity when building the key.

For ciphering a service configuration file you will need a key. Most probably a set of organizational rules will exist, telling you where to store or where to retrieve from such keys. What follows are some general recommendations about what can be done using the CEE.Net's functionality.

Encryption may be required to be done by a proprietary implementation of the abstract service interface `<root namespace>.Base.Abstract.Crypt.Common.etier.Service.IECryptManager`. If you would decide for example to use the service implemented by `<root namespace>.Base.SecurityServices.Crypt.Common.etier.Service.IECryptManager`, you would request the service in this way:

```
String myCryptoService =
    "it.bz.prov.Base.SecurityServices.Crypt.Common.etier.Service.IECryptManager";

IECryptManager cryptoSrv =
    (IECryptManager)broker.GetService(myCryptoService, CommonSourceTypes.LocalOnly);
```

Note

Cryptographic services are domain specific. None of the CEE.Net editions contains a service implementing the `IECryptManager` interface in order not

to "export" security relevant services.

Since cryptographic services change when moving from one environment to another you should guarantee the following:

- Provide a (unencrypted) configuration file for any Business Component (or any tier) which allows you to specify which cryptographic service to use for decrypting security relevant stuff;
- Whenever reading a configuration, check if the file is readable without decrypting (always accept configuration files in clear text); this is important since you will not pass encrypted configuration files to the operations staff, but it will be their tasks to encrypt them later when they will deploy your Business Component, after having set the domain specific parameters as needed.

For more detailed information about key or certificate handling and encryption services to use please refer to the guidelines provided by your employer.

5.12. Explaining the Rules

This chapter explains what the single rules found in chapter 4 are good for. Some exist due to technical requirements, others by architectural or organizational ones.

5.12.1. Organizational Guidelines

O.2: A Business Component's common assembly provides the interfaces of all the services offered by the Business Component and the data types that are going to be exchanged. All this can easily be done by referencing only the service broker's common assembly. As Business Components are self contained and autonomous blocks, they will by definition not depend on any other components.

O.11: Remoteable service interfaces are a prerequisite for distributing Business Components to different servers. It is very likely to happen that your organization won't decide to host all services on one single system.

O.17: The BusinessResult class allows to tell the caller if the operation has been successful or not, or which problems have been encountered. They are richer in functionality than SOAP exceptions and support multiple languages. As the BusinessResult is a structured object, GUI frameworks can provide default viewers for visualizing the qualitative results of a service call (note that this is not the data output of the service call).

O.23: COM interoperability with Business Components consists only of service consumption, a Business Component cannot be written with COM technology. This means that COM programs are nothing more than service clients and are therefore only allowed to access a Business Component's enterprise tier.

O.36: It is essential to have the service interfaces of a Business Component documented in order to leverage parallel development of service tiers and clients.

O.37: It is essential to provide a readable description of a Business Component's service interfaces.

O.38: A compiled HTML file represents a complete reference book for understanding a Business Component's interfaces, including how they work and how they should or should

not be used. This is key to success for decoupling developers from each other with parallel tasks.

O.39: To agree on a tool for building compiled HTML help files is a great deal when you have automation tasks in mind (such as automatic building of the help files), apart from the fact that it contributes to readability and consistency.

O.41: Versioning is important for deployment.

O.100: Any explanation would be pleonastic.

O.101: Any explanation would be pleonastic.

O.102: This rules possible communication paths in order to gain, along with the guidelines that rule inter-Business-Component-communication - a clear integration pattern, coherent with organizational communication paths.

O.103: This rules possible communication paths and guarantees that all levels of a Business Component are involved in data processing (always starting from the level nearest to business).

O.106: One of the key aspects of Business Components is that they are reusable; they call each other exclusively by calling services of each other's enterprise tier. Having a Business Component's enterprise tier services not remoteable means to limit its reusability to components deployed to one single machine.

O.107: This is defined by the reference architecture in order to rule communication paths between Business Components.

O.203: This translates into the architectural rule that a resource tier cannot call neither another tier of its own Business Component nor any other Business Component except such of the Foundation- or the Auxiliary layer.

O.216: Security and privacy are important aspects. Please read the chapter about implementation patterns and guidelines.

O.217: This aims at complete separation of operations and software development as it allows system administrators to move resources without the need of reconfiguring applications and services. Please read the chapter about implementation patterns and guidelines.

O.220: See O.39

O.303: The enterprise tier is the only tier allowed to contact other Business Components which are not of the Foundation- or the Auxiliary layer.

O.316: See O.216

O.319: See O.39

O.403: This translates into the architectural rule that a workspace tier cannot call neither another tier of its own Business Component nor any other Business Component except such of the Foundation- or the Auxiliary layer.

O.415: See O.216

O.418: See O.39

O.503: This translates into the architectural rule that a user tier cannot call neither another tier of its own Business Component nor any other Business Component except such of the Foundation- or the Auxiliary layer.

O.514: See O.216

O.516: See O.39

5.12.2. Technical Rules

T.1: The service broker's common assembly allows the single distributed components of a Business Component to talk to the CEE.Net and to use its functionality. This is essential for participating in SOA.

T.3: Deploying an assembly that references a specific version of another assembly which has not been deployed to the GAC involves problems in upgrade scenarios: since the referencing assembly sticks to a specific version of another assembly, the latter cannot be upgraded without upgrading also the referencing one. This will most probably compromise minor upgrades via auto-deployment.

T.4: This rule aims at readability of source code.

T.5: The CEE.Net Kernel expects all service interfaces – regardless of the tier – to be within a namespace which ends with `.Service`. This is essential in order to improve performance when analyzing the Business Components deployed to the components folder. The other elements of the namespace structure ruled by T.5 aim at readability of source code.

T.6: All the types exchanged between a service and a client must be XML serializable; this is always true for plain (nested) classes with primitive properties and for enumerations, but not necessarily for more complex types.

T.7: see T.6.

T.8: the MUST condition of the rule derives from the fact that the CEE.Net locator and the assembly analyzer expect interface names to start with the uppercase letter `I`. The rest of the rule aims at readability of source code.

T.9: By inheriting from this interface you provide the CEE.Net at runtime with information for effective service handling (such as association of the service with its Business Component or the tier, and much more important, the fact if a service instance can be reused for multiple requests). If you don't, the CEE.Net will nor care about this information at runtime with a possible loss of performance.

T.10: This is useful whenever you either want to extend an existing service interface or if you want your service have to implement a so called abstract service interface for service switching at runtime.

T.12: The CEE.Net's WS publisher does not publish a service if it does not respect this rule. This is in order to allow NOT to publish a more specialized service interface to remote clients although the parent interface is qualified for publishing.

T.13: Properties cannot be published via Web Services.

T.14: The CEE.Net must intervene on these types when enhancing standard web service functionality. It expects the types to be found in either the common assembly containing the service interface or in the service broker's common assembly.

T.15: The CEE.Net must intervene on these types when enhancing standard web service functionality. It can deal with one dimensional arrays, but it can't with multidimensional ones.

T.16: The CEE.Net must intervene on these types when enhancing standard web service functionality. It can deal with one dimensional arrays, but it can't with multidimensional ones. In addition, the web service specification is unclear in respect to multidimensional arrays.

T.18: This adheres to the WS Standard.

T.19: This adheres to the WS Standard.

T.20: This rule is an expression of the CEE.Net's COM interoperability feature.

T.21: This rule is an expression of the CEE.Net's enhanced Java interoperability feature.

T.22: This rule is an expression of the CEE.Net's COM interoperability feature.

T.24: This rule is an expression of the CEE.Net's COM interoperability feature. It is useful when the method is built in a way that is incompatible with COM (such as `in`-arrays not defined as `ref` or types which can't be marshaled automatically).

T.25: This adheres to the Microsoft recommendations and aims at facilitating service consumption with COM clients.

T.26: FDTs are auto-generated by the Business Component designer, and there's no possibility to describe and generate functionality with the BC designer.

T.27: This adheres to the Web Services interoperability standard.

T.28: This adheres to the Web Services interoperability standard.

T.29: This adheres to the Web Services interoperability standard.

T.30: Functionality cannot be XML serialized, only data can.

T.31: This rule is an expression of the CEE.Net's enhanced Java interoperability feature.

T.32: If a method has been declared to be COM interoperable, all the classes used by this specific method either as a return type or as a method parameter must be known to COM.

T.33: This is complementary to T.32.

T.34: This rule addresses an advanced feature of the CEE.Net of complex interface handling: When ignoring T.26 (and thus doing without the features of the Business Designer) different service implementations could hide differently "intelligent" FDTs behind the FTD-interfaces.

T.35: Why not?

T.40: Without the common assembly clients cannot access the Business Component by the CEE.Net.

T.104: This simply says that services of a tier can only be called from a remote machine if they implement the `IRemoteable` interface. Implementing this interface is a prerequisite for having a service published by the CEE.Net.

T.105: user tier services are not XML serializable (they inherit from graphical components) as thus not remoteable.

T.201: The service broker's common assembly provides base service classes, essential types such as `BusinessResult` and methods for requesting services. A service implementation can't do without.

T.202: The service must implement its interface, which is defined in the Business Component's common assembly.

T.204: Why not?

T.205: See T.3.

T.206: See T.4.

T.207: This aims at readability of source code.

T.208: Inheriting from a CEE.Net's base service saves you from a lot of implementation issues, explained in the chapter about implementation patterns.

T.209: This means that a service implementation must implement an interface of the same tier, and it cannot implement more than one (multiple service interfaces are not supported by the CEE.Net's assembly analyzer).

T.210: This adheres to Microsoft's recommendations about COM interoperability.

T.211: This aims at readability of source code.

T.212: This aims at project readability.

T.213: The service broker cannot create an instance of a remote service without a default constructor. Constructor parameters can be passed only to instances of local services.

T.214: This makes a service suitable for being used by the CEE.Net's Kernel by requesting it from the service broker. The "normal" way to request a service instance is to create a service broker in order to ask for a service instance. To accomplish the task, the service broker needs a type library which is going to be requested from the Kernel. When it is the Kernel himself to request a service, this results in a deadlock: while the Kernel waits for the service broker to be created, the latter waits for the Kernel to respond with a type library. The Kernel will avoid this passing the type library directly to the service broker's constructor, so that no type library request is needed. If your service that is finally going to be requested by the Kernel from the service broker internally creates another service broker, the type library has to be propagated to your service in order to be propagated to the internal service broker in order not to end up in a dead lock once again.

T.215: This helps along when analyzing program flows in debug scenarios.

T.218: This expresses the architectural rule that applications (COM interoperability is limited to service consumption) are not allowed to access the resource tier of a Business Component.

T.219: This assembly contains the implementation of the Business Component's resource tier and has to be deployed to the CEE.Net's components folder. Please refer to the chapter about packaging for further details.

T.301: See T.201

T.302: See T.202

T.304: See T.204

T.305: See T.205

T.306: See T.206

T.307: See T.207

T.308: See T.208

T.309: See T.209

T.310: See T.210

T.311: See T.211

T.312: See T.212

T.313: See T.213

T.314: See T.214

T.315: See T.215

T.317: This expresses the architectural rule that applications (COM interoperability is limited to service consumption) are allowed to access the enterprise tier of a Business Component.

T.318: This assembly contains the implementation of the Business Component's enterprise tier and has to be deployed to the CEE.Net's components folder. Please refer to the chapter about packaging for further details.

T.401: See T.201

T.402: See T.202

T.404: See T.204

T.405: See T.205

T.406: See T.206

T.407: See T.207

T.408: See T.208

T.409: See T.209

T.410: See T.210

T.411: See T.211

T.412: See T.212

T.413: See T.213

T.414: See T.215

T.416: This expresses the architectural rule that applications (COM interoperability is limited to service consumption) are allowed to access the workspace tier of a Business Component.

T.417: This assembly contains the implementation of the Business Component's workspace tier and has to be deployed to the CEE.Net's components folder. Please refer to the chapter about packaging for further details.

T.501: See T.201

T.502: See T.202

T.504: See T.204

T.505: See T.205

T.506: See T.206

T.507: See T.207

T.508: See T.208

T.509: See T.209

T.510: See T.211

T.511: See T.212

T.512: See T.213

T.513: See T.215

T.514: This assembly contains the implementation of the Business Component's user tier and has to be deployed to the CEE.Net's components folder. Please refer to the chapter about packaging for further details.

T.601: This is an integration issue: All common assemblies must end up in the root of the CEE.Net's components folder in order to allow Business Components which are deployed and called locally to call each other. If you decide to deploy a Business Component's common assembly to a subfolder of the CEE.Net's components folder, only Business Components deployed side by side yours will be able to contact it. By putting your common assembly into a subfolder you are going to isolate your Business Component from others.

T.602: This aims at structural clarity. Please read the related note in the chapter about packaging.

T.603: It depends on the fact whether a packaging format is feature based or not when deciding if to create one single package or multiple ones. The intention is to have the choice to install on a server only a part of a Business Component. Either you install the part as a feature of a multi-featured package (such as Windows Installer package could be), or you create multiple packages, one per part.

T.604: This is because other Business Components deployed to the same CEE.Net need the common assembly of your Business Component in order to contact it. If you are going to distribute your Business Component, you have to deploy its common assembly redundantly since it entirely describes your component.

T.605: This is because of the assembly probing algorithm of the .Net framework.

T.606: Developers need some more information about your Business Component, provided by the developer package. Especially the description of service interfaces is crucial, and IntelliSense is just advantageous.

T.607: ZIP archives are not feature based. See T.603.

T.608: Deploying only the common assembly to a CEE.Net can make sense whenever the CEE.Net has only to publish service interfaces to a registry without offering a service implementation.

T.609: See T.606