CEE.Net, BC Designer and UI-Frameworks

Conceptional Features

Ver. 1.0

Official Documentation

CEE.Net, BC Designer and UI-Frameworks - Ver. 1.0

Conceptional Features

04. Sep. 2009

Official Documentation

Author:

Zeno Moriggl

Describes the CEE.Net in team with BC Designer and SharpForge

Content

1. Abstract

This document is about the CEE.Net working in team with the BC Designer and the SharpForge Framework. It explains what the CEE.Net is, describes the conceptual features it offers and explains how both the BC Designer and the SharpForge framework act as a facilitators for perfect intergration with the CEE.Net infrastructure.

2. What is the CEE.Net?

The CEE.Net is a lightweight infrastructural component whose compound of instances represents a ubiquitous base for hosting and accessing bundled services. From a technical point of view, all the CEE.Net instances realize a SOA infrastructure which, though profiting from and built on leading technologies, on the one hand applies a corset to service development, enforcing to be respected a set of architectural rules, and on the other hand leverages parallel service development by supporting the contract first approach in service design.

The benefit from introducing rules and enforcing them by a infrastructural component such as the CEE.Net is to have the ability to hide them from developers and to move them to the infrastructure. Instead of having to verify again and again if the set of architectural rules, aimed to guarantee such important concepts as seamless interoperability, security, site awareness and so on, has been respected, you can trust your infrastructure to do it for you.

The loop gets closed if considering not only the CEE.Net but also the facilitators (BC Designer and BC Generator) and the frameworks that seamlessly connect you to your SOA world.

The following chapter describes the features of the CEE.Net that have been implemented in addition to the functionality natively offered by the .Net Framework on which it has been built. Each feature explains why it has the right to exist.

2.1. Features

2.1.1. Transparent Proxy

The CEE.Net implements a concept of transparent proxy for service clients, hiding from the developer if the service is offered by a remote server or by the local machine. From a technical point of view, the CEE.Net allows two ways to access a service: either via Web Services if the service is offered by a remote computer, or via a dynamically loaded local class instance if the service is offered by the local machine. This speeds up local services enormously.

2.1.1.1. Why do we need this?

This feature brings the developer a seamless SOA experience: he has not to care about local or remote services; all he has to do is to request a service. This requires less programming skills and lowers costs of software development.

For the operating staff transparent proxying allows to distribute bundled services on different servers without braking functionality of other services or applications and without any need of reconfiguring applications. This lowers costs of operating, increases agility and contributes to complete decoupling of Development and Operations.

2.1.2. Integrated Security via Kerberos

The CEE.Net natively integrates with Kerberos security realms when calling remote services, following a configurable pattern for dynamic SPN resolving and supporting Kerberos delegation.

2.1.2.1. Why do we need this?

No discussion: services have to authenticate the calling user before responding, that's a strong and indispensable security requirement.

Furthermore, this feature allows seamless integration with the user's current security context as long as he has been authenticated by a KDC (this is, in our case, Active Directory authentication) and brings to the user a complete SSO experience, saving him from remembering a lot of passwords which potentially end up on post-its or other improper support media.

The feature is implemented unrestrictedly relying on the open WSS standards (OASIS) and guarantees interoperability with Java-based services, independent from whether they are running on a Windows, a Unix or a Linux system. This is important for not restricting the SOA infrastructure to a single platform, an important criterion for at least theoretical vendor independence.

All what we have said above can be accomplished by simply using the core functionality of the .Net framework. So what's the advantage of the feature?

Obviously, the feature is built on the core functionality already present in the .Net framework. What the feature does is to move the implementation away from where it normally happens: the service client or the service implementation. It's no longer up to the developer of the service or the developer of the client to care about requesting and integrating the Kerberos token and configuring the WS policy for having matching what the service expects to get and what the client sends. This work is done exclusively by the CEE.Net, hiding it completely from the developer, by introducing a dynamic SPN resolving pattern for requesting the tokens (otherwise we would have to control that all developers respect the rules for SPN naming for having all the stuff working).

The feature is described in detail by the document *SOA Infrastructure Security.pdf*.

This, once again, requires less programming skills and lowers costs of software development.

2.1.3. Client Data Forwarding

The CEE.Net can automatically propagate infrastructural data about the client to a remote service. This includes two properties which are protected by the CEE.Net in a special way and represent the user's identity:

- The name of the user calling the service from the client
- The name of the computer from where the service is called

Furthermore, the CEE.Net can propagate a configurable set of additional properties which are controlled by the single application calling the remote service and not by the CEE.Net itself. This could, for example, be the name of the office the user is currently working for.

2.1.3.1. Why do we need this?

This feature is important for two reasons:

First, in our environment it is crucial to know on the one hand *who is working where* in order to control site aware access to redundantly deployed resources (such as replicated databases). Basing on the user's physical location we have to find out *which* resources (file shares, printers, databases and so on) to access. On the other hand, we must know *whom the user is working for*, since users can belong to multiple offices and can play different roles for different offices. Along wit the user's identity, this is a prerequisite for *controlling access* to the resources that have earlier been found to be physically available at his current position.

Second, we have to propagate the user's identity, and may be additional client data, between service hops and – more important – between different security contexts in order to provide the destination service with the necessary data for guaranteeing controlled access to his resources, basing on the user's identity. This means that if a client calls a remote service which on his part, in order to carry out what he has been requested to do, must call another remote service, the identity and other fundamental client data of the original caller must be propagated by the first remote service to the second one (which on his part will propagate it to the third one and so on).

One could object that standard approaches to fulfill these requirements already did exist in the .Net framework and had not to be implemented once again. Indeed, Kerberos delegation would do great part of the job, but:

- Letting a Web Server impersonate a user's identity requires the user account to have elevated privileges and possibly access to the resources that the service must deal with;
- Kerberos delegation ends where the Kerberos realm ends and could therefore be used in the own Intranet; but how should identities be forwarded from one security context to another?
- The Kerberos token does not contain additional information such as for whom the user is currently working.

When searching for an alternative way for transmitting the original user's identity and other fundamental data one could suggest to put them in all methods of all services that are going to be created. Moving this feature to the CEE infrastructure saves us from the duty to control if this rule (which would have to be much more precise) has been respected, and it saves the developer from the obligation to not forget about its correct and – as we are talking about identities – reliable implementation.

The protected properties of the feature (username and computername) are described in detail by the document *SOA Infrastructure Security.pdf*.

2.1.4. Gateway Functionality

The CEE.Net has the ability to dynamically understand if a service that has been requested is inside or outside his security realm. Dependently from what the CEE.Net has found out, he decides how to authenticate against the remote service: if he finds that the service is an intranet service, a Kerberos token is integrated in the SOAP header, as described above (with all the advantages of SSO). If he finds out that the service is an external one, the CEE.Net will create a Username token with username and password, trying to request them from a gateway service he has been told to use, and if no one are returned, he will ask the user to provide them interactively.

2.1.4.1. Why do we need this?

This feature is needed for basic message based intra-realm authentication without having to rely on superior certification authorities. The feature helps to realize a trust relationship between different organizations which authenticate against each other and trust each other to authenticate the own users in a correct way. Identities are forwarded from one organization to another by the feature *Client Data Forwarding* described above.

This feature is described in detail by the document *SOA Infrastructure Security.pdf*.

2.1.5. Autodeployment

The CEE.Net has the ability to automatically synchronize its components folder (this is the container hosting the artifacts of the Business Components) with one or more update folders. New or updated artifacts are deployed without stopping the CEE.Net (components currently in use by clients are marked and then disposed as soon as they get released by the consumer; contemporaneously they are substituted with the new release).

2.1.5.1. Why do we need this?

We need this feature, which is aimed to facilitate daily operations, for guaranteeing an agile method for deploying new or updated Business Components.

2.1.6. Lightweight Distributed Infrastructure

The CEE.Net installed in the production environment is currently about 1,5 MB of size and available in two versions: the Client Edition which is lacking of capabilities for service publishing for remote access, and the Server Edition which allows to publish the services of locally hosted Business Components to the rest of the world. A core functionality of all CEE.Net editions is site awareness or, in other words, the ability to access services nearby depending on a underlying physical network partitioning. So if services are deployed redundantly, the closest instance can be found dynamically.

These characteristics make the CEE.Net suitable for being installed on all computers and if needed on servers without a great performance and resource impact.

2.1.6.1. Why do we need this?

Except the core administration of the Autonomous Province of Bolzano, which is concentrated in a quite well connected area at Bolzano, a lot of peripheral branch offices exist, which are mainly thought to service the citizen where he lives. In the same way as services such as teaching, road network maintenance, district survey of any type or other things that usefully have to happen in a redundant way, near the people, are offered nowadays in single districts, an electronic service equivalent may be needed. This is especially the case when the necessary connectivity for an acceptable user experience along with an acceptable reliability cannot be guaranteed. Therefore the SOA infrastructure must support redundantly deployed Business Components which all offer the same services but limit them to their service area.

In distributed environments a lightweight, easy to deploy and easy to manage infrastructure is crucial. If the components to install for having the SOA running are too heavy, distribution will rarely happen with all the negative impacts such as loss of performance and poor user experience.

2.1.7. Logical Component Clustering

Logical component clustering of the CEE.Net allows to subdivide a SOA environment into logical subparts offering to the world a subset of the organization's Business Components. Different clusters can contain the same Business Components, though they may differ in version or behavior.

While the service broker (this is the part of the CEE.Net that connects an application dynamically to a service) can access multiple, prioritized logical clusters contemporaneously, a single CEE.Net Server instance can publish its services to exactly one logical cluster only.

All editions of the CEE.Net can implicitly create logical clusters which match the well connected network segment they are running in (site), or they can implicitly subdivide the site into more than one logical clusters.

2.1.7.1. Why do we need this?

We need logical component clustering for the following reasons:

- To have running multiple versions of a Business Component in a given environment (such as the production environment); this can be extremely useful in migration scenarios when upgrading to new versions of a Business Component
- To speed up dynamic service lookup in distributed environments, if tightly coupling logical component clustering with physical network segmentation (which is an implicit function of the CEE.Net); this saves the service broker from doing a lot of work when he has to filter out the nearest service from a list of hundreds of servers all offering a requested service.

2.1.8. Java/.Net Interoperability

The CEE.Net enhances interoperability between Java Web Services and .Net Web Services which is natively guaranteed by the WS Standards. What the CEE.Net does is handling of nullable data types in .Net service interfaces: the incompatibility between the Java and the .Net platform is given by the fact that some data types such as Date or DateTime natively are primitive and therefore not nullable in the one platform, while they are nullable in the other. So while one of the partners can deal with (send or receive) nulls for class instances of the affected types, the other can't and will end up in runtime errors. The CEE.Net allows to use the .Net's generics pattern in order to declare explicitly in the service interface those datatypes which natively are not nullable, as nullable. In order to have the stuff running, the service broker has to modify at runtime the web service proxy which has been created dynamically from the service's WSDL: the WSDL does not tell anything about nullable or not nullable data types, and the class inferred from the web service description would therefore not compile if forced to implement the service interface whose data types have been enriched with the nullable attribute. The service broker modifies the proxy generated from the WSDL in order to have it matching the service interface in the common assembly.

Note

This feature conflicts with the feature of COM Interoperability!

2.1.8.1. Why do we need this?

We need this feature for seamless integration between services written on the Java platform and such written in .Net. Without this feature Java developers would have for example to know what .Net's MinDate or MaxDate are, and we would have to convey on a rule that says that a .Net's MinDate means the absence of information instead of being a real date.

2.1.9. COM Interoperability

All editions of the CEE.Net automatically register to COM both the interface of the CEE's service broker and all the interfaces of services belonging to locally deployed Business

Components which have been marked as COM visible. If you design a Business Component with the Business Component Designer, all service interfaces of enterprise tier services are automatically declared to be COM visible. The CEE.Net handles also the lifecycle of his COM registered components: if an already registered component gets its interfaces updated, a COM re-registration will take place, and if a Business Component is removed from the CEE.Net, its COM registration will be undone.

Note

This feature conflicts with the feature of Java/.Net Interoperability!

2.1.9.1. Why do we need this?

COM registration of the CE.Net itself and of the service interfaces he offers opens the SOA world to all applications which can deal with COM, including all the infrastructural features that the CEE.Net brings to developers and administrators. Although one could force vendors to develop new solutions on either the .Net or the Java platform, existing commercial products nowadays are still built on other platforms, and especially if they are exceeding a critical size, rewriting an alternative will no longer be convenient. With COM interoperability such applications can participate in SOA, although in a unilateral way only: they can only consume services without offering anything to the rest of the world.

2.1.10. Dynamic Endpoint Discovery

All CEE.Net editions can look up dynamically at runtime all the possible endpoints of a given service interface that is going to be requested, making use of one or more pluggable service registry connectors. The service registry can be of any type, e.g. LDAP, UDDI, a relational database or whatever (obviously, for a specific service registry a fitting registry connector has to be deployed to the CEE.Net for having it attached dynamically at runtime). The CEE.Net supports multiple registries and tries them all at runtime till one returns an endpoint for what he is searching for.

The CEE.Net server edition feeds exactly one service registry by the means of the appropriate, pluggable, registry connector, publishing the endpoints of those services that are marked as remotable and which have successfully been delivered to the Web Server.

2.1.10.1. Why do we need this?

Dynamic endpoint discovery and publishing is vital for severing ones ties with physical deployment. This feature brings to administrators the agility they need for an effective resource management or in migration scenarios. They do not have to deal with tons of server- and client-side configuration files when moving a component from one server to another for performance or maintenance reasons, or when distributing a single Business Component to multiple machines. This is a crucial feature for decoupling an autonomous operations staff from development, as intended by ITIL.

2.1.11. Service Implementation Switching

The CEE.Net supports interface inheritance when coupling interfaces with classes that implement them. This allows a client to request different implementations for one single service interface.

2.1.11.1. Why do we need this?

This feature has been introduced for two reasons:

- In outsourcing scenarios, domain specific services have to be simulated by alternative, domain independent service implementations (such as services that provide the roles of the currently logged on user, or the offices he belongs to, or a service that encrypts a configuration file with domain specific keys …). This feature saves us from recompiling the solution for having it to stick to the domain specific services instead of the simulating ones (and the alternative to name the simulating services in the same way as the domain specific ones is quite dangerous since they can't coexist).

- In migration scenarios, this feature allows a temporary coexistence of two or more concurrent providers for one single service. Let's say tomorrow you decide to no longer want to use Active Directory as your attribute provider. You will have thousands of client instances stuck to the service that currently connects you to Active Directory. With this feature you are not forced to guarantee that all of them can immediately switch to the new attribute provider because you have no other possibility than to substitute the existing service implementation with the new one. This feature will help you especially in distributed environments where changes propagate slowly, due to the replication schedules.

2.1.12. Pluggable Base Components

The CEE.Net introduces the concept of so called base services. Here we are talking about Business Components that offer services that are vital either for the CEE.Net itself or for frameworks in order to accomplish their infrastructural tasks such as:

- Physical network segmentation in order to operate site aware
- Role based server discovery for finding registry servers dynamically
- Kerberos Information services
- Cryptographic tasks
- Providing of credentials for external services (gateway services)
- Information about the user's environment (who he is, which offices he is belonging to, whom he is currently working for, where he is, which resources he has near him, which roles does he play …)

The Business Components that offer this functionality are regular Business Components that can be called also by any other client whenever they want; what qualifies them for Base Components is that they are used *also* by the CEE.Net itself and by UI-frameworks.

An important characteristic of Base Components is that they are domain specific. Network segmentation for example will be differ from domain to domain not only at a logical level, but also in implementation: the system describing the network may be Sites and Services of Active Directory or anything else. The same can be said for cryptographic services, attribute providers, server roles, credential management and so on.

If Base Components are missing, the CEE.Net will lose the features relying on the missing services.

In order to have the ability to simulate ones environment in outsourcing scenarios, the CEE.Net comes natively with a limited set of Base Components suitable for simulating what in the production system will then be substituted with a domain specific Base Component. This includes network segmentation, information about the user's environment and credentials for external services.

This feature is built on the feature of *Service Implementation Switching*.

2.1.12.1. Why do we need this?

We need this feature in order to

- attach dynamically to any of our environments for site awareness, access control, authentication and so on (either production or test) without having to recompile
- simulate our environment in outsourcing scenarios in order to allow a vendor to develop its solution in its own company
- have the ability to donate the CEE.Net to partners without revealing any security relevant details of our environment, and to give them the possibility to profit from the CEE.Net's features by sticking to their own environment, saving them from doing a lot of configuration.

2.1.13. Contract first

The CEE.Net facilitates a contract first approach (along with the Business Component Designer) by attaching at runtime a so called common assembly to the Web Service proxy dynamically created from its WSDL file. This common assembly is created by the Business Component designer when the interface of a Business Component is designed and can immediately be given away to the developer who intends to consume the service. It provides him with all he needs to access the service programmatically, although the service may not even have an implementation yet. At runtime, the CEE.Net attaches to the service by reading its WSDL and creating a proxy dynamically at runtime, and coupling at the end the latter with the common assembly which has been used by the client to access the service long before the WSDL did exist.

2.1.13.1. Why do we need this?

We need this feature for parallel development of a service and its consumer: once designed the service contract, both the service developer and the client developer can start with their work.

2.1.14. Service Publishing

The CEE.Net can automatically publish all or part of the services of locally deployed Business Components to its local IIS instance for having them accessible from remote machines. This feature includes the features 2.1.15, 2.1.16 and 2.1.17.

2.1.14.1. Why do we need this?

This feature saves the developer from having to deal with method decorating and XML serialization tasks. He does not have to know anything at all about Web Services. This requires less programming skills and lowers costs of software development.

2.1.15. Service Access Policies and Service Injection

If a service is offered via Web Service, the CEE.Net Server Edition creates a little wrapper which internally calls the locally deployed service destined to be offered to remote machines. This wrapper is decorated with all the attributes needed to having it published automatically by IIS, and furthermore it can do some infrastructural access control when a remote client is calling the service. This includes to:

- check if a valid security token has been provided via the SOAP header
- check to whom the security token has been issued, if dealing with a Kerberos token

- check if the client has tried to pretend to be someone else than shown by the security token, although not allowed to do so
- inject the internal service instance with the authentication and client data propagated via the SOAP header

2.1.15.1. Why do we need this?

This feature is needed to improve security: although the Web Server can (and should) be told to check if the SOAP header contains a security token, the service wrappers created by the CEE.Net Server Edition do not know if the Web Server really did carry out this task (due to erroneous IIS configuration for example). In order to keep under control security, the wrapper can do the check once again.

Furthermore, this feature is essential for allowing identity- or role -based access control at the inside of the local service instance called by the wrapper. This access control is no longer infrastructural, but explicitly implemented in the service itself. Without this feature the identity of both the originally calling user and the authenticating user would be unknown to the internal service instance. Actually, this is true also for the user's position and additional client data (such as for whom he is currently working). All these attributes may be needed for an advanced access control, since a user may have access to resources owing to his position, or he may play different roles for different employers, and the same role may imply different permissions for different employers.

In addition, this feature is a prerequisite for forwarding of client data between service hops since it allows to check if the authenticating user is allowed to forward other identities than himself. Without this feature, if a remote service has to call yet another remote service, the latter would be deprived of the ability to control access to its resources basing on the user's data. The aspect of service hopping is described in detail in the *SOA Infrastructure Security.pdf* document.

2.1.16. Complex Interface Handling

The Web Service standard lacks a lot of features that OO-developers in the meantime have become fond of. The intent to realize a transparent proxy in a SOA environment implies to adhere to the standards used for remote service calls when designing a service interface, and as a consequence to do without all the features of the object oriented approach that are in contrast to these standards. This includes:

- Interface data types exchanged between methods of different classes
- Method overloads
- Bidirectional object dependency in hierarchical trees (objects pointing to their parents)
- Exchange of objects not fully decomposable into primitive data types

The WS publisher of the CEE.Net Server Edition on the one hand and the service broker on the other can hide some of these limitations from the developer, by dynamically handling method overloads (expanding them at server side to multiple methods and collapsing them client side to a method overloaded multiple times); or by substituting at server side interface types in method parameters with the data types implementing them, and reverting to interfaces at the client side after deserialization; or by cutting dynamically references to parent objects in method parameters in order to allow XML serialization.

2.1.16.1. Why do we need this?

We need this feature in order to improve the developer experience, but we could also do without it since in most cases we pay for it with loss of performance.

The feature is more important in so called abstract interfaces when decoupling from a concrete service implementation (see feature 2.1.16) and for COM interoperability since COM needs a interface for its data types (see feature 2.1.9).

2.1.17.Specialized Types Serialization

The WS-publisher of the CEE.Net Server Edition automatically includes specialized data types in XML serialization even when they do not explicitly appear in any of the methods of a Business Component's services. Specialized data types means such inheriting directly or indirectly from any data type that is either used as a method parameter or a return value of a service method. If the WS-publisher would not interfere when creating the service wrapper, such data types would not be serializable and could not be used in remote service calls.

2.1.17.1.Why do we need this?

We need this feature in order to build effective service interfaces relying on type specialization rather than on flat, flag- and property overloaded objects, nevertheless without inflating unnecessarily the numbers of methods. Let's say you have a service that does all the bureaucratic stuff for marrying two people, and assume the procedure differs depending on their provenience, may be requiring other documents for foreigners. In this case one could think of the following alternatives:

Alternative 1

- a method `MarryWithForeigner(Person person, Foreigner foreigner);`
- a method `Marry(Person person, Person person);`
- a method `Marry(Foreigner foreigner, Foreigner foreigner);`

Alternative 2

- A method `Marry(Person person, Person person);`

  with inflation of the `Person` data type, decorating it with all the properties needed for describing a foreigner which will not be used in most cases

Alternatice 3

- A method `Marry(Person person, Person person);`

  which accepts both a `Person` object and a `Foreigner` object as long as inheriting from a `Person`.

The advantages of alternative 3 are quite obvious:

While alternative 1 every time when a new person type has going to be supported adds a new method to the service – and last but not least – one for each combination of person types when couples (or even worse, n-tuples) of person interact, alternative 3 can deal with one single method.

If we take a look at alternative 2, we have an analogous situation: even though we have not to increase the number of the methods, we have to inflate the person data type. This alternative is even worse than the first one for two reasons: first, if you were not able to think just from the beginning of all the possible person types you will have to deal with, you will run in changes that will break backward compatibility of the service interface (or you end up with adding yet another method) by breaking the service contract. Second, you will have to explain which properties of the Person object are important and which to ignore under which circumstances. Since there will be a set of properties specific to a specific person type, some properties will always be null. To deduce from the object's properties of which type a person is can be a defying task, especially if you have to understand which properties are absolutely needed for describing a person type completely.

With alternative 3 we have the chance to build much more stable service interfaces, subject to much less changes, and this is key to success in SOA.

2.1.18.Service Preferences

The CEE.Net's service broker allows a service consumer to specify some preferences such as if to prefer local services or remote ones, or if to connect to remote services only when they are in the client's side, or if to request services only if published to a specific service registry.

2.1.18.1.Why do we need this?

This feature is important for giving the analysts the possibility to build different scenarios when dealing with offline capabilities of applications or different deployment scenarios.

3. What Is the BC Designer?

The BC Designer is a graphical tool for designing a Business Component. Along with its code generator it allows rapid BC development and guarantees compatibility with the CEE.Net's development patterns.

The BC Designer allows to describe the bundle of service interfaces offered by a Business Component and stores this description in an XML format compatible with the one used by its Java equivalent (which is realized as an Eclipse plug in). By agreeing to share a common description languages, independently from which platform specific designer has been used to describe a Business Component, each platform can pick up such a description in order to produce all the stuff needed for immediately docking to the component's services (this is, the service interfaces itself and all the classes exchanged in service calls). The .Net Generator will obviously produce a .Net assembly containing the service interfaces and the data type classes, while the Eclipse plug in will produce a jar file. On both platforms, the produced outputs are ready for use in software projects and constitute an essential prerequisite for the contract first approach. This is what the BC Designer's code generator produces for the service consumers.

In addition to the service interfaces and the data types involved in method calls (compiled to a single assembly which will be referenced by whoever wants to access the Business Component), the BC Designer's code generator produces an "empty" service implementation, consisting of as many Visual Studio projects as tiers have been designed within the Business Component. Each project is complete of all classes and methods needed for implementing the component's services, and each project will compile immediately after its generation. All the developer has to do is to implement the still empty methods of the single services. Having done this, the output files are ready for being deployed to a CEE.Net. This is what the BC Designer's code generator does for who implements the service (instead of using it).

4. What Is SharpForge?

SharpForge is a UI framework which completes the trio of supportive tools for rapid application development. From its large feature list, the following are infrastructural and go hand in hand with or are influenced by the SAO approach:

- **Seamless service access by the means of an integrated service handler**: Every application basing on the SharpForge framework automatically connects to the local CEE.Net (which is a prerequisite for SharpForge). All SharpForge pages can request services through the framework's service handler without the need to directly bind to the CEE.Net. In addition, the framework handles service broker instances saving the developer from dealing with performance issues deriving from inconvenient implementation patterns.
- **Native integration with the user's environment and cryptography**: Every SharpForge application supports dynamic resource discovery (such as database servers) and identifies key user data (such as his offices or roles) by binding to predefined abstract service interfaces offered by the underlying CEE.Net. Due to the fact that the service interfaces are abstract, the application can easily switch between service implementations and has therefore the ability to dock dynamically to different environments. In this way any SharpForge application operates site aware (basing on network segmentation specific to the environment where it is running), can cipher its configuration files with organization specific algorithms and allows role based access control basing on organization specific authentication systems.
- **Task- and process driven approach**: SharpForge is conceptually close to SOA as it thinks in tasks and processes: it rather executes operations (calls methods) than operating on data tables. It is focusing on navigation between steps as elementary building blocks of tasks.