



BONNIE.NET WEB EDITION

VERSION 4.0.3.1 USER GUIDE

This page intentionally left blank.



This guide explains the functionalities added to Bonnie.NET Standard Edition by Bonnie.NET Web Edition.

For this reason, knowledge of Bonnie.NET Standard Edition User Guide is required.



This page intentionally left blank.

SUMMARY

SUMMARY	3
VERSION 4.0.3.1 NEW FEATURES.....	5
INTRODUCTION.....	7
BONNIE.NET WEB EDITION ARCHITECTURE	9
SIGNERBUTTONS	11
<i>SignerButton Control</i>	12
<i>Properties</i>	12
<i>Methods</i>	14
<i>Events</i>	15
<i>StringSignerButton Control</i>	16
<i>TextSignerButton Control</i>	16
<i>StreamSignerButton Control</i>	17
<i>FileSignerButton Control</i>	18
<i>CustomSignerButton Control</i>	19
<i>Using the EventArgs</i>	20
<i>Using Ajax with SignerButtons</i>	21
CASSANDRA CLIENT FRAMEWORK.....	25
PKCS7SIGNER	27

SIGNATUREBROWSERS	29
<i>Pkcs7SignatureInfo</i>	31
ENDNOTES.....	35
<i>Localization</i>	35
<i>Search engines Optimization</i>	36

VERSION 4.0.3.1 NEW FEATURES

The 4.0.3.1 version of Bonnie.NET Web Edition adds to the previous release the following new features:

Upgrade to the .NET Framework 4.0 of the entire library.

Bonnie.NET Web Edition has been update to the new .NET Framework 4.0. The CAS architecture has been revisited in term of the new Level2 Security Transparence model.

CAPICOM ActiveX has been removed.

The dependency by CAPICOM ActiveX has been removed. The same has been substitute by Cassandra Client Framework. This permit to (1) removes the most part of the yellow bars that appears on Internet Explorer when CAPICOM is activated, (2) use *SignerButtons* even in Internet Explore x64, (3) sign files from client without having to submit them to the server first, (4) select the encoding to use when text is signed (5) upgrade over time Bonnie.NET Web Edition with new functionality without having to wait for the improvement of CAPICOM ActiveX.

Added the possibility to use Ajax calls to the server rather than standard postback.

SignerButtons can now submit asynchronously to the server the signature computed using the *XmlHttpRequest* object. Developers can specify a custom JavaScript function to run before the

Ajax submission to the server and a custom JavaScript function to be run after the callback to the client.

Added the new FileSignerButton control.

The *FileSignerButton* control has been added to the collection of *SignerButtons*. While the old *StreamSignerButton* is now used only to sign stream of data that come from the server, the new *FileSignerButton* permits to sign files that reside on the client machine.

Added the possibility to select the encoding to use for the signature of texts.

Developers can now choose the encoding to use when texts are signed using the *Pkcs7Signer* or the *SignerButtons*.

FileSignatureBrowser can now directly save to the file system the signed file.

The *FileSignatureBrowser* implements a new method, *SaveTo(...)*, that permits to save on the file system the file contained on the signed-data message.

INTRODUCTION

Bonnie.NET Web Edition adds to Bonnie.NET Standard Edition a set of functionalities that permit to generate or exchange signed data by using the PKCS#7 standard.

Bonnie.NET Web Edition allows to:

- Exchange signed data by using a wide adopted format (the PKCS#7 standard) that can be parsed by all the cryptographic systems that use it (interoperability).
- Encapsulate, in a single message (a PKCS#7 signed-data message), 1) the signature computed, 2) the signed data, 3) a description about the signature, 4) useful attributes such as the signature date and time, the digital certificate of the signer and 5) its identity. All of them are signed too.
- Generate multiple signature of the same piece of data. The signed-data message can contain an unlimited number of signatures generated by different signers, each one with its signature date and time, its description, the public key of the signer and its identity.
- Browse PKCS#7 signed-data messages to get their content, signatures date and time, identities of the signers and other useful information.
- Perform the signed-data message generation both from desktop application than from ASP.NET web pages (both server side than client side).
- Verify signatures contained on a single PKCS#7 signed-data message.



This page intentionally left blank.

BONNIE.NET WEB EDITION ARCHITECTURE

Bonnie.NET Web Edition respect to the Standard Edition:

- introduces a new crypto-object: *Pkcs7Signer* crypto-object: It allows the generation of a PKCS#7 signed-data message starting form a *X509Signer* crypto-object.
- Introduces four ASP.NET web controls, *TextSignerButton*, *StreamSignerButton*, *FileSignerButton* and *CustomSignerButton*. They permit to generate, within web pages, signatures of texts, streams, files or custom data, using X.509 digital certificates stored on the client machine or on a smart card connected to it. The signature is then sent to the server as PKCS#7 signed-data message base64 encoded.
- Introduces a set of SignatureBrowsers *objects* that permit to browse PKCS#7 signed-data messages getting their content and their signatures information.

While the *Pkcs7Signer* operate on desktop applications or on the server side of ASP.NET applications, the *SignerButtons* work on the client side within the user's browser. The signature on the client is made possible by using the [Cassandra Client Framework](#) that has to be installed on the client machine. This component operates on the user's browser and has two main functions: (1) to select a certificate from the user's certificates store, (2) to generate a PKCS#7 signed-data message starting from the certificate selected and the user's data. The message generated is then sent to the server through a post back of the page or through an AJAX submission.

User's data can be anything: 1) texts written by the user inside a page's input control, 2) texts sent by the server to the user and contained inside some page's input control, 3) files selected

by the user and that resides on its machine, 4) streams generated or stored on the server and sent to the user or 5) custom data obtained by executing a custom JavaScript function.

The four controls derive from the base *SignerButton* control. It encapsulates all the features related to the signature process. The *SignerButton* control supports multiple signatures. When some data is signed by a user, the same can be re-signed by other users.

When using *SignerButtons*, the signed-data message generated will contain 1) signatures values, 2) signed data, 3) public keys¹ of the signers, 4) signing date and time (in UTC format), 5) document name if the signature was performed on a file and 6), optionally, descriptions about each signature.

With *SignatureBrowsers*, the signed-data message generated with *Pkcs7Signer* or with *SignerButtons* (or that came from other sources) can be parsed and the data inside it can be extracted and used on the business process.

¹ The X.509 digital certificate without the private key.

SIGNERBUTTONS

As seen, *SignerButtons* are a specific set of HTML input buttons that can be used for signature generation from an ASP.NET web page. They are divided in:

TextSignerButton: It permits to generate the signature, as PKCS#7 signed-data message, of some text contained on some page's input control and to deliver it to the server as base64 string.

StreamSignerButton: same as the previous but related to the signature of streams (for example files) that came from the server.

FileSignerButton: It permits the PKCS#7 signed-data message generation of files stored on the user's machine.

CustomSignerButton: It uses a custom JavaScript function to obtain a string to sign. The JavaScript function can rearrange data inside the web page in any type of custom text (for example as xml document) that is then signed.

The four controls derive from the **SignerButton** abstract class. It implements common method regarding controls usage and behavior. The *TextSignerButton* and the *CustomSignerButton* derives from the base **StringSignerButton**. It permits to select the encoding to use to generate the signature of texts.

The derivation tree is given by:

SignerButton

└ *StringSignerButton*

└ *TextSignerButton*

└ *CustomSignerButton*

└ *StreamSignerButton*

└ *FileSignerButton*

SIGNERBUTTON CONTROL

The *SignerButton* control derives from the *System.Web.UI.WebControls.WebControl* class and adds to it the properties, methods and events described below.

PROPERTIES

The *SignerButton* class implements the following properties:

Name	Type	Description
ButtonText	string	Text that will be displayed on the Button surface.
ButtonStyle	ButtonStyle	Style of the button as <i>ButtonStyle</i> enum value. The <i>SignerButton</i> can be rendered as standard Button, LinkButton or ImageButton.
ImageButtonSrc	string	Uri of the image to use when the button is rendered as ImageButton
ImageButtonDisabledSrc	string	Uri of the image to use when the button is rendered as ImageButton and the Button is disabled.
PostBackExceptions	bool	If true (default), exceptions are posted back to the server. If false, when an exception

		occurs, the page is not posted back.
NotSupportedBrowserAction	BrowserAction	Action to perform when the button is rendered on browsers different from Internet Explorer.
CertificateSelectDialogTitle	string	Title of the dialog box that appears on the client when a request to select a certificate is made.
CertificateSelectDialogMessage	string	Message that will be printed on the dialog box that appears on the client when a request to select a certificate is made.
UseAjax	bool	If true, the <i>SignerButton</i> performs an Ajax submission to the server rather than a normal postback.
PreGenerateScriptFunction	string	Invocation string for a JavaScript function that will be executed before the signature generation. If it return false, the procedure ends without generating the signature.
AjaxSendScriptFunction	string	Invocation string for a JavaScript function that will be executed before the Ajax submission to the server. If it return false, the submission to the server does not occur and the procedure ends.
AjaxCompleteScriptFunction	string	Invocation string for a JavaScript function that will be executed after the AJAX callback from the server.
FrameworkNotFoundScriptFunction	string	Invocation string of a JavaScript function that will be called if Cassandra Client Framework is not found on the client. It will replace the default JavaScript function implemented inside <i>SignerButtons</i> .
EnableSeo	bool	If true, the <i>SignerButton</i> rearrange itself to

		meet the crawler requirements.
UpgradeFrameworkScriptFunction	string	For future use.

METHODS

The *SignerButton* class implements the following methods:

Name	Description
SetControlForDescription(Control control)	Set the control that will be used by the user to insert some comment during the signature generation. The comment will be stored on the signature.
SetControlForException(Control control)	Set the control that will be used to display exception messages on the client. This is used only if <i>PostBackExceptions</i> is false.
SetClientExceptionDescription (SignatureExceptionReason exception, string exceptionMessage)	Set, for each exception defined by the <i>SignatureExceptionReason</i> enum, the custom exception message to display on the client when <i>PostBackExceptions</i> is false and an exception occurs.
GetClientExceptionDescription (SignatureExceptionReason exception)	Get the default exception message defined for each exception identified by the <i>SignatureExceptionReason</i> enum
DoCoSignature(string signedData)	It permits to put the <i>SignerButton</i> in a state that co-signatures are generated. It accepts as argument a signed-data message previous generated. By pressing the <i>SignerButton</i> , the signed-data message will be co-signed by the new user. NB: the signed-data message value is not

maintained on the page round-trip and so the method must be called on each post back. This was done to avoid unwanted data (that, with stream or file signatures, can have large dimension) to be maintained on the page.

EVENTS

The *SignerButton* class implements the following event:

Name	Description
OnSignatureDone	This event is fired when the signature generated is sent to the server.

The *OnSignatureDone* event is defined as:

```
public event EventHandler<SignatureDoneEventArgs> OnSignatureDone;
```

The *SignatureDoneEventArgs* object has the following methods and properties

Name	Type	Description
SignedMessage	string	It represents the signed-data message returned by the client.
HasException	bool	True if an exception occurs on the client during the signature generation.
ClientException	SignerButtonClientException	Contains information about the exception that eventually occurs on the client.

STRINGSIGNERBUTTON CONTROL

The *StringSignerButton* class is an abstract class that extends the *SignerButton* control in order to allow the selection of the encoding to use when texts are signed.

It adds to the *SignerButton* class the following property:

Name	Type	Description
TextEncoding	TextEncoding	Enum for the encoding to use for the signature of texts. Its default value is UTF8.

TEXTSIGNERBUTTON CONTROL

The *TextSignerButton* control extends the *StringSignerButton* control in order to allow the signature of some text contained on a web page's input control. Text can be inserted by the user or can be sent to it by the server.

It adds to the *StreamSignerButton* class the following method:

Name	Description
SetControlForSignature(Control control)	It permits to set the control that contain/will contain the text to sign.

The following example shows how to set up a *TextSignerButton* control (*btSign*) for the signature generation of texts contained in a *TextBox* control named *myText*. Optionally, a description can be inserted by the user in a *TextBox* named *myDescription*:

```
//set the control that contain/will contain the text to sign  
btSign.SetControlForSignature(myText);
```

```
//set the control to use to insert some description on the signature
btSign.SetControlForDescription(myDescription);

//set the event handler
btSign.OnSignatureDone += new
    EventHandler<SignatureDoneEventArgs>(btSign_OnSignatureDone);
```

We will see in the following paragraphs how the event handler works.

Remember that the two previous methods don't maintain the state during the postback of the page and so they must be called on each page load.

STREAMSIGNERBUTTON CONTROL

The *StreamSignerButton* control extends the *SignerButton* control in order to allow the signature of streams of data. The most common case is given by files that reside on the server.

It adds to the *SignerButton* class the following method and property:

Name	Description
<code>SetStreamForSignature (Stream stream, string streamName, bool closeStream)</code>	Set the stream that must be signed. Optionally, a stream name (for example a file name if the stream is a file) can be passed as argument. In this case the <i>streamName</i> value will be signed too. If <i>closeStream</i> is true, the stream will be closed after the signature.
<code>string StreamName</code>	Get or Set the name of the stream that will be signed with the stream.

After the signature, the stream's content is contained on the signed-data message and it can be recovered from the same (see the *SignatureBrowsers* on the next chapter).

The following example shows how to set up a *StreamSignerButton* control (*btSign*) for the signature generation of a stream (named *myStream*). Optionally, the user can insert a description for the signature in the *TextBox* named *myDescription*. The same will be signed during the signature process.

```
//set the stream to sign and the related stream name
btSign.SetStreamForSignature(myStream, "myStream");

//set the control to use to insert some description of the signature
btSign.SetControlForDescription(myDescription);

//set the event handler
btSign.OnSignatureDone += new
    EventHandler<SignatureDoneEventArgs>(btSign_OnSignatureDone);
```

FILESIGNERBUTTON CONTROL

The *FileSignerButton* control extends the *SignerButton* control in order to allow the signature of files stored on the user's machine. When the *SignerButton* is pressed, a file open dialog box asking the user to select the file to sign will be displayed on the client.

The *FileSignerButton* permits to set the filter to use on the dialog box, giving to developers the ability to prevent the selection, by the user, of unwanted files types.

The *FileSignerButton* class adds to the *SignerButton* class the following properties:

Name	Type	Description
AllowedExtension	AllowedExtension	Enum for the allowed files extensions of the files to sign. Values can be combined with a bitwise OR operation.

CustomExtensions	ProtectedList<string>	List of all the files extensions not covered by the AllowedExtension enum but that are permitted.
------------------	-----------------------	---

After the signature, the file will be contained, as stream, inside the signed-data message and it can be recovered from the same (see the *SignatureBrowsers* on the next chapter).

The following example shows how to set up a *FileSignerButton* control (*btSign*) that permits to sign files that resides on the user's machine. In this example, files can be: 1) Pdf documents, 2) Word documents or 3) custom .cust files. Optionally, a description can be inserted by the user in a TextBox named *myDescription*. This will be signed too:

```
//set the allowed extensions
btSign.AllowedExtension = AllowedExtension.Doc | AllowedExtension.Pdf;

//set the custom extensions
btSign.CustomExtensions.Add(".cust");

//set the control to use to insert some description on the signature
btSign.SetControlForDescription(myDescription);

//set the event handler
btSign.OnSignatureDone += new
    EventHandler<SignatureDoneEventArgs>(btSign_OnSignatureDone);
```

CUSTOMSIGNERBUTTON CONTROL

The *CustomSignerButton* control extends the *TextSignerButton* control in order to allow the signature of a custom message obtained by executing a JavaScript function on the client. The function must return a string value that will be signed.

It adds to the *TextSignerButton* class the following property:

Name	Type	Description
CustomScriptFunction	string	Set the function invocation string to use for the generation of the custom text to sign.

The following example shows how to set up a *CustomSignerButton* control (*btSign*) for the signature generation of a string generated by a JavaScript function named *myFunction*. Optionally, a description can be inserted by the user in a TextBox named *myDescription*. The same will be signed during the signature process.

```
//set the function invocation string
btSign.ScriptFunction = "myFunction(param1,param2...)";

//set the control to use to insert some description on the signature
btSign.SetControlForDescription(myDescription);

//set the event handler
btSign.OnSignatureDone += new
    EventHandler<SignatureDoneEventArgs>(btSign_OnSignatureDone);
```

USING THE EVENTARGS

In the previous examples, we used the *btSign_OnSignatureDone* method as argument for the event handler. A typical implementation of the method should be:

```
// Event handler for the button
void btSign_OnSignatureDone(object sender, SignatureDoneEventArgs e)
{
    //if the client generates and exception, the event can manage it
    if (e.HasException)
    {
        switch (e.ClientException.SignatureExceptionReason)
        {
```

```
        {  
            case SignatureExceptionReason.DataNotFound:  
                ManageException(e.ClientException.Message);  
                break;  
            case SignatureExceptionReason.DataException:  
                ...  
                ...  
                break;  
        }  
        return;  
    }  
  
    //save in some place the signature generated  
    SaveSignatureValue(e.SignedMessage);  
}
```

First of all, the method checks if an exception occurs. If so, it is managed, else, the signed-data message is saved in some place.

USING AJAX WITH SIGNERBUTTONS

SignerButtons can use Ajax rather than a standard post back to submit to the server the signature computed on the client. To do so, the *UseAjax* property must be set to *true*. In this way, after the generation of the signature, the same is sent to the server asynchronously.

Developers can specify a JavaScript function to run before the submission of the signature to the server and a JavaScript function to run after the callback to the client. The two could be used to perform some visual effect such as the display of a progress bar or an animated gif.

To specify a JavaScript function to run before the computed signature submission, use the *AjaxSendScriptFunction* property:

```
//set the function invocation string to run before the submission to  
//the server  
btSign.AjaxSendScriptFunction = "myFunction(param1,param2...)";
```

If you want to prevent the submission to the server of the signature generated in response to a particular condition that occurs during the JavaScript function execution, use the `return false` statement inside the function. In this case, the submission does not occur and the procedure ends.

To specify a JavaScript function to run after the signature generation, use the `AjaxCompleteScriptFunction` property:

```
//set the function invocation string to run after the callback to
//the client
btSign.AjaxCompleteScriptFunction = "myFunction(param1,param2...)";
```

With this second property, `SignerButtons` will add an extra argument, a JavaScript object, at the end of the arguments list. This object will contain the signatures data.

The following example shows how an `AjaxCompleteScriptFunction` can be made. The server side of the web page must contain the following instruction:

```
//set the function invocation string to run after the callback to
//the client
btSign.AjaxCompleteScriptFunction = "myTestFunction(param1)";
```

In this example, `btSign` is a `SignerButton` and `myTestFunction` is a JavaScript function invocation string that accepts a single argument.

The function could be something like this:

```
//JavaScript function to run after the ajax response
function myTestFunction (param1, o) {

    //if the added object is defined
    if (typeof (o) !== 'undefined') {
        //if exception
        if (o.status === 'err') {

            // ... actions to perform if an exception occurs
        }
    }
}
```

```

    }
    else {
        for (var i = 0; i < o.signatures.length; i++)

            // ... actions to perform for each signature

        }
    }
}

```

Notice that, while the server side code doesn't include the *o* parameter for the value of the *AjaxCompleteScriptFunction* property, the *myTestFunction* function need to specify it to allow the access to the same.

If the signature generates an exception, the JavaScript object will contain two properties:

Name	Value	Description
status	err	It states that the signature process generated an exception
message		Exception message related to the exception thrown.

If the signature process ends successfully, the object will contain the following properties:

Name	Value	Description
status	ok	It states that the signature process ends successfully.
signatures		Array with all the signatures contained on the signed-data message generated.

The *signatures* array will contain, as items, a list of objects, one for each signature contained on the signed-data message. The single object has the following properties:

Name	Description
signer	Common name of the certificate of the signer.
date	Date and time, as local date and time, at which the signature was computed.

description	Description related to the signature.
-------------	---------------------------------------

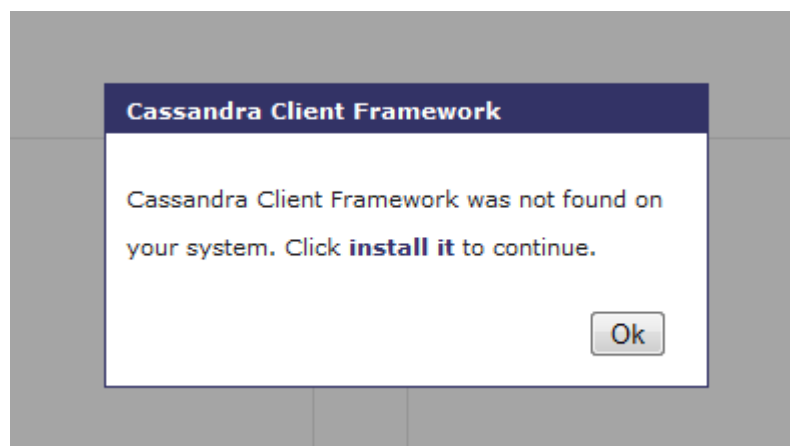
We recall that, with Ajax, the web page's user interface cannot be modified server side. The *OnSignatureDone* event can still be used to execute some code on the server (as, for example, to save the signature computed in some place) but if the server side code try to modify the web page's user interface, the same will not be refreshed. Updates must occur on the client side during the execution of the *AjaxCompleteScriptFunction* function.

CASSANDRA CLIENT FRAMEWORK

As seen, the signature with *SignerButtons* from a web browser is made possible by using the [Cassandra Client Framework](#) that must be installed on the client. It consists on a .NET assembly that works on the browser as an ActiveX component.

Developers have not the need to worry about the installation of Cassandra Client Framework on each client. In fact, *SignerButtons* are able to detect if the same is already installed and, if it not so, they prompt the user to install it.

By default, a modal dialog appears on the user's browser:



By clicking the *install it* link, the user will download the Cassandra Client Framework setup from we-coffee.com site and the installation begins. If needed, the .NET Framework 4.0 Client Profile is installed too.

Developers can modify the default behavior in two ways:

1. By modifying the default message using the *SetClientExceptionDescription(...)* method of the *SignerButton* control using, as *SignatureExceptionReason*, the value *FrameworkNotInstalled*. For example:

```
myButton.SetClientExceptionDescription  
  
    (SignatureExceptionReason.FrameworkNotInstalled, "myCustomText");
```

2. By replacing the default JavaScript function that created the modal dialog with a custom JavaScript function. To do so, the *FrameworkNotFoundScriptFunction* property must be used. For example:

```
myButton.FrameworkNotFoundScriptFunction = "myCustomFunction()";
```

In the previous code, *myButton* is a *SignerButton* and *myCustomFunction()* is the invocation string related to the *myCustomFunction* JavaScript function.

PKCS7SIGNER

The *Pkcs7Signer* is a class that permits to generate PKCS#7 signed-data messages of texts, bytes arrays, files and all type of streams from desktop application or server side code.

It encapsulates the *X509Signer* crypto-object² in order to obtain a PKCS#7 signed-data message rather than a standard signature.

It implements the same *Sign(...)* and *SignStream(...)* methods of the *X509Signer* class but it adds to them some overloads that permits to:

1. Set a description for the signature to generate.
2. Set a stream name if a stream is signed.
3. Set the file name if a file is signed.

It implements moreover the following additional methods:

Name	Description
CoSign(string signedData)	Permit the co-signature of a previous generated signature.
Verify(string signedData)	Permit to verify a PKCS#7 signed-data message.

With *Pkcs7Signer*, the signature verification is now computed with a common method (the *Verify(...)* method) for all type of signatures. This because all the information needed to validate a signature (substantially, the X.509 certificate) is stored in the PKCS#7 signed-data message itself and the *X509Signer* object is not involved in the verification process, whatever type of signature generated.

² For the *X509Signer* crypto-object see the Bonnie.NET Standard User Guide

The following example shows how to use the *Pkcs7Signer* crypto-object to obtain a signed-data message of a demo text using an X.509 digital certificate whose friendly name is given by “*mycertificate*”.

```
//create an X509Signer object
X509Signer x509 = X509Signer.Create("mycertificate");

//encapsulate the X509Signer object in a Pkcs7Signer object
Pkcs7Signer signer = Pkcs7Signer.Create(x509);

//create the signed-data message
string signedData = signer.Sign("demo");
```

SIGNATURE BROWSERS

SignatureBrowsers are special objects that permit to parse a signed-data message base64 encoded and extract from it all its information.

SignatureBrowsers implemented on Bonnie.NET Web Edition are:

Pkcs7SignatureBrowser: base class for the browsing of PKCS#7 signed-data messages.

Pkcs7StringSignatureBrowser: class for the browsing of PKCS#7 signed-data messages involving signatures of texts.

Pkcs7StreamSignatureBrowser: class for the browsing of PKCS#7 signed-data messages involving signatures of streams.

Pkcs7FileSignatureBrowser: class for the browsing of PKCS#7 signed-data messages involving signatures of files.

The derivation tree is given by:

Pkcs7SignatureBrowser

└ *Pkcs7StringSignatureBrowser*

└ *Pkcs7StreamSignatureBrowser*

└ *Pkcs7FileSignatureBrowser*

Their constructors need, as argument, the PKCS#7 signed-data message³ previously generated. They permit to explore it by using the following properties:

Name	Type	Description
SignatureInfoCollection	All	Collection of Pkcs7SignatureInfo objects.
Signature	All	Get or set the signed-data message, base64 encoded.
SignedString	Pkcs7StringSignatureBrowser only	Get the text that was signed.
Encoding	Pkcs7StringSignatureBrowser only	Encoding used to create the signature of texts. The default value is UTF8.
SignedStream	Pkcs7StreamSignatureBrowser and Pkcs7FileSignatureBrowser only	Get a read-only stream that permits to extract from the signed-data message the signed content.
SignedFileName	Pkcs7FileSignatureBrowser only	Name of the file (or stream) that was signed.

SignatureBrowsers permit to verify signatures contained on the signed-data message by using the following method:

Name	Description
VerifySignature()	Permits to verify the signature contained on a PKCS#7 signed-data message.

³ Being the PKCS#7 signed-data message format a standard, Pkcs7SignatureBrowsers are able to browse not only signatures computed with Bonnie.NET Web Edition but all other signatures computed by others cryptographic systems that make use of the same standard.

As general rule, you must use:

- *Pkcs7StringSignatureBrowser* when you want to access the signature's content as string.
- *Pkcs7StreamSignatureBrowser* when you want to access the signature's content as stream (it can be a string, a stream or a file)
- *Pkcs7FileSignatureBrowser* when you want to access the signature's content as stream and get the stream name or the file name that was set during the signature. The stream can be saved to disk by using the following method:

```
myBrowser.SaveTo(@"c:\...\myFileName.xxx");
```

where the argument is an arbitrary path to the file system.

PKCS7SIGNATUREINFO

SignatureBrowsers store all the information related to a single signature in a **Pkcs7SignatureInfo** object.

SignatureBrowsers expose, as property, a collection of these objects. For each signature contained on the PKCS#7 signed-data message, a *Pkcs7SignatureInfo* object is created and then added to the collection.

The *Pkcs7SignatureInfo* class implements the following properties:

Name	Type	Description
Certificate	X509Certificate2	The certificate the signed the content.
UTCSignatureDate	DateTime	Date and time in which the signature was generated expressed as UTC format.

SignatureDescription ⁴	string	Description added to the signature by the signer.
SignerIdentityInfo	SignerIdentityInfo	Special version of the Distinguished Name of the signer.

The following example shows how to browse a signed-data message generated with *SignerButtons*.

As seen, the event handler of *SignerButtons* uses the *SignatureDoneEventArgs* to store the signed-data message computed on the client. Being *e* this object, and supposing that a text was signed:

```
//create a Pkcs7StringSignatureBrowser object passing
//the signed-data message to the constructor
Pkcs7StringSignatureBrowser browser =
    new Pkcs7StringSignatureBrowser(e.SignedMessage);

//write to the response stream the signed (original) text
Response.Write("Signed Message: " + browser.SignedString + "<br/>");

//write to the response stream the information related to each signature
foreach (Pkcs7SignatureInfo i in browser.SignatureInfoCollection)
{
    //write the common name of the signer
    Response.Write("message signed by : "
        + i.SignerIdentityInfo.CommonName + "<br/>");

    //convert the UTC date time in a local date time
    DateTime d = i.UTCSignatureDate.ToLocalTime();
```

⁴ For compatibility with CAPICOM, the signature description is given by a signed attribute with OID given by 1.3.6.1.4.1.311.88.2.2.



```
//write the signature date
Response.Write(" on: " + d.ToShortDateString()
              + " - " + d.ToShortTimeString() + "<br/>");

//write the signature description
Response.Write("Description: " + i.SignatureDescription + "<br/>");
}
```



This page intentionally left blank.

ENDNOTES

LOCALIZATION

Generally, Bonnie.NET doesn't make use of resources that need to be localized.

The Bonnie.NET Web Edition, however, permits to manage on the client all the exceptions that can occur during the signature process.

As we know, in case of exceptions, *SignerButtons* can display a message to the user directly on the client without making the post back to the server.

In this case, messages to display are contained on the Bonnie.NET Web Edition assembly. To override their values (maybe in another language) you can use the following method of *SignerButtons* objects:

Name	Description
SetClientExceptionDescription (SignatureExceptionReason reason, string exceptionMessage)	For a given <i>SignatureExceptionReason</i> , set the exception message to display on the client (if <i>PostBackExceptions</i> is false)

This method accepts as arguments a *SignatureExceptionReason* enum value and a string. The first is used to select the exception description that must be overridden while the second is used to set the message that will override the default value. For example, being *btSign* a *SignerButton* control, you can write this line of code:

```
btSign.SetClientExceptionDescription  
    (SignatureException.DataNotFound, "No data to sign was found.");
```

If the *SignerButton* does not find any data to sign, the "No data to sign was found." message will be displayed on the control used as argument for the *SetControlForException(...)* method.

SEARCH ENGINES OPTIMIZATION

Despite the simplicity of their usage, *SignerButtons* generate complex content on a web page. When search engines scan the page, sometimes they can find obstacles on the parsing of the same, generating the HTTP 500 internal server error. To avoid it, *SignerButtons* implement the following property:

Name	Type	Description
EnableSeo	bool	Enable the search engines optimization for the <i>SignerButton</i> .

When set to true, if the *SignerButton* detects that the page is under a search engines crawling, it will render itself as normal HTML input control, without inserting on the page all the functionalities needed to perform digital signatures.