## Creating Internet server applications: Overview

Topic groups  See also

Delphi allows you to create Web server applications as CGI applications or dynamic-link libraries (DLLs). These Web server applications can contain any nonvisual component. Special components on the Internet palette page make it easy to create event handlers that are associated with a specific Uniform Resource Identifier (URI) and, when processing is complete, to programmatically construct HTML documents and transfer them to the client.

In order to create Web server applications, it is important to understand

- Terminology and standards
- HTTP server activity
- Delphi Web server applications
- Debugging server applications

**Note:** You can also use ActiveForms as Internet server applications. For more information about ActiveForms, see Generating an ActiveX control based on a VCL form.

## Terminology and standards

Topic groups  See also

Many of the protocols that control activity on the Internet are defined in Request for Comment (RFC) documents that are created, updated, and maintained by the Internet Engineering Task Force (IETF), the protocol engineering and development arm of the Internet. There are several important RFCs that you will find useful when writing Internet applications:

- RFC822, “Standard for the format of ARPA Internet text messages,” describes the structure and content of message headers.
- RFC1521, “MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies,” describes the method used to encapsulate and transport multipart and multiformat messages.
- RFC1945, “Hypertext Transfer Protocol — HTTP/1.0,” describes a transfer mechanism used to distribute collaborative hypermedia documents.

The IETF maintains a library of the RFCs on their Web site, www.ietf.cnri.reston.va.us

These documents include, among other information, details about

- Parts of a Uniform Resource Locator
- HTTP request header information
- HTTP server activity

## Parts of a Uniform Resource Locator

Topic groups  See also

The Uniform Resource Locator (URL) is a complete description of the location of a resource that is available over the net. It is composed of several parts that may be accessed by an application. These parts are illustrated in the following figure:

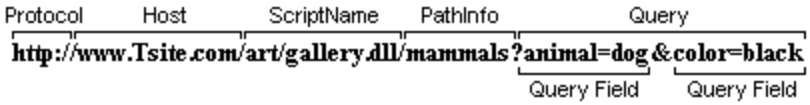
The first portion (not technically part of the URL) identifies the protocol (http). This portion can specify other protocols such as https (secure http), ftp, and so on.

The Host portion identifies the machine that runs the Web server and Web server application. Although it is not shown in the preceding picture, this portion can override the port that receives messages. Usually, there is no need to specify a port, because the port number is implied by the protocol.

The ScriptName portion specifies the name of the Web server application. This is the application to which the Web server passes messages.

Following the script name is the pathinfo. This identifies the destination of the message within the Web server application. Path info values may refer to directories on the host machine, the names of components that respond to specific messages, or any other mechanism the Web server application uses to divide the processing of incoming messages.

The Query portion contains a set a named values. These values and their names are defined by the Web server application.




### URI vs. URL

The URL is a subset of the Uniform Resource Identifier (URI) defined in the HTTP standard, RFC1945. Web server applications frequently produce content from many sources where the final result does not reside in a particular location, but is created as necessary. URIs can describe resources that are not location-specific.

## HTTP request header information

Topic groups See also

HTTP request messages contain many headers that describe information about the client, the target of the request, the way the request should be handled, and any content sent with the request. Each header is identified by a name, such as “Host” followed by a string value. For example, consider the following HTTP request:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

The first line identifies the request as a GET. A GET request message asks the Web server application to return the content associated with the URI that follows the word GET (in this case /art/gallery.dll/animals?animal=doc&color=black). The last part of the first line indicates that the client is using the HTTP 1.0 standard.

The second line is the Connection header, and indicates that the connection should not be closed once the request is serviced. The third line is the User-Agent header, and provides information about the program generating the request. The next line is the Host header, and provides the Host name and port on the server that is contacted to form the connection. The final line is the Accept header, which lists the media types the client can accept as valid responses.

## HTTP server activity

Topic groups  See also

The client/server nature of Web browsers is deceptively simple. To most users, retrieving information on the World Wide Web is a simple procedure: click on a link, and the information appears on the screen. More knowledgeable users have some understanding of the nature of HTML syntax and the client/server nature of the protocols used. This is usually sufficient for the production of simple, page-oriented Web site content. Authors of more complex Web pages have a wide variety of options to automate the collection and presentation of information using HTML.

Before building a Web server application, it is useful to understand how the client issues a request and how the server responds to client requests:

- Composing client requests
- Serving client requests
- Responding to client requests

## Composing client requests

Topic groups  See also

When an HTML hypertext link is selected (or the user otherwise specifies a URL), the browser collects information about the protocol, the specified domain, the path to the information, the date and time, the operating environment, the browser itself, and other content information. It then composes a request.

For example, to display a page of images based on criteria selected by clicking buttons on a form, the client might construct this URL:

```
http://www.TSite.com/art/gallery.dll/animals?animal=dog&color=black
```

which specifies an HTTP server in the www.TSite.com domain. The client contacts www.TSite.com, connects to the HTTP server, and passes it a request. The request might look something like this:

```
GET /art/gallery.dll/animals?animal=dog&color=black HTTP/1.0
Connection: Keep-Alive
User-Agent: Mozilla/3.0b4Gold (WinNT; I)
Host: www.TSite.com:1024
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg, */*
```

## Serving client requests

Topic groups  See also

The Web server receives a client request and can perform any number of actions, based on its configuration. If the server is configured to recognize the /gallery.dll portion of the request as a program, it passes information about the request to that program. The way information about the request is passed to the program depends on the type of Web server application:

- If the program is a Common Gateway Interface (CGI) program, the server passes the information contained in the request directly to the CGI program. The server waits while the program executes. When the CGI program exits, it passes the content directly back to the server.
- If the program is WinCGI, the server opens a file and writes out the request information. It then executes the Win-CGI program, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. The server waits while the program executes. When the program exits, the server reads the data from the content file written by the Win-CGI program.
- If the program is a dynamic-link library (DLL), the server loads the DLL (if necessary) and passes the information contained in the request to the DLL as a structure. The server waits while the program executes. When the DLL exits, it passes the content directly back to the server.

In all cases, the program acts on the request of and performs actions specified by the programmer: accessing databases, doing simple table lookups or calculations, constructing or selecting HTML documents, and so on.

## Responding to client requests

Topic groups  See also

When a Web server application finishes with a client request, it constructs a page of HTML code or other MIME content, and passes it back (via the server) to the client for display. The way the response is sent also differs based on the type of program:

- When a Win-CGI script finishes it constructs a page of HTML, writes it to a file, writes any response information to another file, and passes the locations of both files back to the server. The server opens both files and passes the HTML page back to the client.
- When a DLL finishes, it passes the HTML page and any response information directly back to the server, which passes them back to the client.

Creating a Web server application as a DLL reduces system load and resource use by reducing the number of processes and disk accesses necessary to service an individual request.

## Web server applications

Topic groups  See also

Web server applications extend the functionality and capability of existing Web servers. The Web server application receives HTTP request messages from the Web server, performs any actions requested in those messages, and formulates responses that it passes back to the Web server. Any operation that you can perform with a Delphi application can be incorporated into a Web server application.

To build a Web server application, follow these steps:

1. Choose one of the types of Web server applications
2. In the IDE, begin Creating the Web Server application
3. Add components following the structure of a Web server application
4. Debug you application, using the techniques described in Debugging server applications

## Types of Web server applications

Topic groups See also

Using the internet components, you can create four types of Web server applications. Each type uses a type-specific descendant of *TWebApplication*, *TWebRequest*, and *TWebResponse*:

| Application Type | Application Object | Request Object | Response Object |
|---|---|---|---|
| Microsoft Server DLL (ISAPI) | *TISAPIApplication* | *TISAPIRequest* | *TISAPIResponse* |
| Netscape Server DLL (NSAPI) | *TISAPIApplication* | *TISAPIRequest* | *TISAPIResponse* |
| Console CGI application | *TCGIApplication* | *TCGIRequest* | *TCGIResponse* |
| Windows CGI application | *TCGIApplication* | *TWinCGIRequest* | *TWinCGIResponse* |

### ISAPI and NSAPI

An ISAPI or NSAPI Web server application is a DLL that is loaded by the Web server. Client request information is passed to the DLL as a structure and evaluated by *TISAPIApplication*, which creates *TISAPIRequest* and *TISAPIResponse* objects. Each request message is automatically handled in a separate execution thread.

### CGI stand-alone

A CGI stand-alone Web server application is a console application that receives client request information on standard input and passes the results back to the server on standard output. This data is evaluated by *TCGIApplication*, which creates *TCGIRequest* and *TCGIResponse* objects. Each request message is handled by a separate instance of the application.

### Win-CGI stand-alone

A Win-CGI stand-alone Web server application is a Windows application that receives client request information from a configuration settings (INI) file written by the server and writes the results to a file that the server passes back to the client. The INI file is evaluated by *TCGIApplication*, which creates *TWinCGIRequest* and *TWinCGIResponse* objects. Each request message is handled by a separate instance of the application.

## Creating Web server applications

Topic groups  See also

All new Web server applications are created by selecting File|New from the menu of the main window and selecting Web Server Application in the New Items dialog. A dialog box appears, where you can select one of the Web server application types:

- ISAPI and NSAPI: Selecting this type of application sets up your project as a DLL with the exported methods expected by the Web server. It adds the library header to the project file and the required entries to the uses list and exports clause of the project file.
- CGI stand-alone: Selecting this type of application sets up your project as a console application and adds the required entries to the uses clause of the project file.
- Win-CGI stand-alone: Selecting this type of application sets up your project as a Windows application and adds the required entries to the uses clause of the project file.

Choose the type of Web Server Application that communicates with the type of Web Server your application will use. This creates a new project configured to use Internet components. It includes

- The Web module
- The Web application object

## The Web module

Topic groups  See also

The Web module (*TWebModule*) is a descendant of *TDataModule* and may be used in the same way: to provide centralized control for business rules and non-visual components in the Web application.

Add any content producers that your application uses to generate response messages. These can be the built-in content producers such as *TPageProducer*, *TDataSetPageProducer*, *TDataSetTableProducer*, *TQueryTableProducer* and *TMIDASPageProducer*, or descendants of *TCustomContentProducer* that you have written yourself. If your application generates response messages that include material drawn from databases, you can add data access components or special components for writing a Web server that acts as a client in a MIDAS application.

In addition to storing non-visual components and business rules, the Web module also acts as a Web dispatcher, matching incoming HTTP request messages to action items that generate the responses to those requests.

You may have a data module already that is set up with many of the non-visual components and business rules that you want to use in your Web application. You can replace the Web module with your pre-existing data module. Simply delete the automatically generated Web module and replace it with your data module. Then, add a *TWebDispatcher* component to your data module, so that it can dispatch request messages to action items, the way a Web module can. If you want to change the way action items are chosen to respond to incoming HTTP request messages, derive a new dispatcher component from *TCustomWebDispatcher*, and add that to the data module instead.

Your project can contain only one dispatcher. This can either be the Web module that is automatically generated when you create the project, or the *TWebDispatcher* component that you add to a data module that replaces the Web module. If a second data module containing a dispatcher is created during execution, the Web server application generates a runtime error.

**Note:** The Web module that you set up at design time is actually a template. In ISAPI and NSAPI applications, each request message spawns a separate thread, and separate instances of the Web module and its contents are created dynamically for each thread.

**Warning:** The Web module in a DLL-based Web server application is cached for later reuse to increase response time. The state of the dispatcher and its action list is not reinitialized between requests. Enabling or disabling action items during execution may cause unexpected results when that module is used for subsequent client requests.

## The Web Application object

Topic groups  See also

The project that is set up for your Web application contains a global variable named *Application*. *Application* is a descendant of *TWebApplication* (either *TISAPIApplication* or *TCGIApplication*) that is appropriate to the type of application you are creating. It runs in response to HTTP request messages received by the Web server.

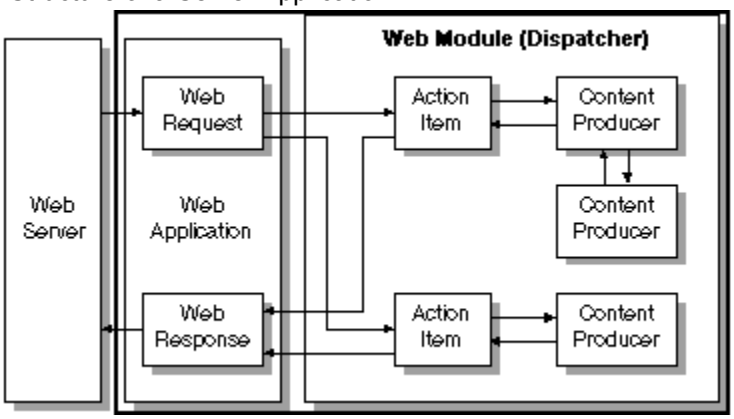
**Warning:** Do not include the forms unit in the project **uses** clause after the CGIApp or ISAPIApp unit. Forms also declares a global variable named *Application*, and if it appears after the CGIApp or ISAPIApp unit, *Application* will be initialized to an object of the wrong type.

## The structure of a Web server application

Topic groups  See also

When the Web application receives an HTTP request message, it creates a *TWebRequest* object to represent the HTTP request message, and a *TWebResponse* object to represent the response that should be returned. The application then passes these objects to the Web dispatcher (either the Web module or a *TWebDispatcher* component).

Structure of a Server Application




The Web dispatcher controls the flow of the Web server application. The dispatcher maintains a collection of action items (*TWebActionItem*) that know how to handle certain types of HTTP request messages. The dispatcher identifies the appropriate action items or auto-dispatching components to handle the HTTP request message, and passes the request and response objects to the identified handler so that it can perform any requested actions or formulate a response message. The action items are responsible for reading the request and assembling a response message. Specialized content producer components aid the action items in dynamically generating the content of response messages, which can include custom HTML code or other MIME content. The content producers can make use of other content producers or descendants of *THTMLTagAttributes*, to help them create the content of the response message.

If you are creating the Web Client in a multi-tiered database application, your Web server application may include additional, auto-dispatching components that represent database information encoded in XML and database manipulation classes encoded in javascript. The dispatcher calls on these auto-dispatching components to handle the request message after it has tried all of its action items.

When all action items (or auto-dispatching components) have finished creating the response by filling out the *TWebResponse* object, the dispatcher passes the result back to the Web application. The application sends the response on to the client via the Web server.

## The Web dispatcher

Topic groups See also

If you are using a Web module, it acts as a Web dispatcher. If you are using a pre-existing data module, you must add a single dispatcher component (*TWebDispatcher*) to that data module. The dispatcher maintains a collection of action items that know how to handle certain kinds of request messages. When the Web application passes a request object and a response object to the dispatcher, it is responsible for dispatching the request message.

Set up the web dispatcher by adding actions to the dispatcher.

## Adding actions to the dispatcher

Topic groups  See also

Open the action editor from the Object Inspector by clicking the ellipsis on the *Actions* property of the dispatcher. Action items can be added to the dispatcher by clicking the Add button in the action editor.

Add actions to the dispatcher to respond to different request methods or target URIs. You can set up your action items in a variety of ways. You can start with action items that preprocess requests, and end with a default action that checks whether the response is complete and either sends the response or returns an error code. Or, you can add a separate action item for every type of request, where each action item completely handles the request.

## Dispatching request messages

Topic groups See also

When the dispatcher receives the client request, it generates a *BeforeDispatch* event. This provides your application with a chance to preprocess the request message before it is seen by any of the action items.

Next, the dispatcher looks through its list of action items for one that matches the pathinfo portion of the request message's target URL and that can provide the service specified as the method of the request message. It does this by comparing the *PathInfo* and *MethodType* properties of the *TWebRequest* object with the properties of the same name on the action item.

When the dispatcher finds an appropriate action item, it causes that action item to fire. When the action item fires, it does one of the following:

- Fills in the response content and sends the response or signals that the request is completely handled.
- Adds to the response and then allows other action items to complete the job.
- Defers the request to other action items.

After checking all its action items, if the message is not handled the dispatcher checks any specially registered auto-dispatching components that do not use action items. These components are specific to multi-tiered database applications, which are described in Building Web applications using InternetExpress

If, after checking all the action items and any specially registered auto-dispatching components, the request message has still not been fully handled, the dispatcher calls the default action item. The default action item does not need to match either the target URL or the method of the request.

If the dispatcher reaches the end of the action list (including the default action, if any) and no actions have been triggered, nothing is passed back to the server. The server simply drops the connection to the client.

If the request is handled by the action items, the dispatcher generates an *AfterDispatch* event. This provides a final opportunity for your application to check the response that was generated, and make any last minute changes.

## Action items

Topic groups  See also

Each action item (*TWebActionItem*) performs a specific task in response to a given type of request message.

Action items can completely respond to a request or perform part of the response and allow other action items to complete the job. Action items can send the HTTP response message for the request, or simply set up part of the response for other action items to complete. If a response is completed by the action items but not sent, the Web server application sends the response message.

Set up action items for your Web server application by

- Adding actions to the dispatcher
- Determining when action items fire
- Responding to request messages with action items

## Specifying action item properties

Topic groups  See also

Most properties of the action item determine when the dispatcher selects it to handle an HTTP request message. To set the properties of an action item, you must first bring up the action editor: select the *Actions* property of the dispatcher in the Object Inspector and click on the ellipsis. When an action is selected in the action editor, its properties can be modified in the Object Inspector.

The action item includes properties that specify

- The target URL
- The request method type

Other properties that influence when the dispatcher fires an action item are described in

- Enabling and disabling action items
- Choosing a default action item

## The target URL

Topic groups  See also

The dispatcher compares the *PathInfo* property of an action item to the *PathInfo* of the request message. The value of this property should be the path information portion of the URL for all requests that the action item is prepared to handle. For example, given this URL,

```
http://www.TSite.com/art/gallery.dll/mammals?animal=dog&color=black
```

and assuming that the /gallery.dll part indicates the Web server application, the path information portion is

```
/mammals
```

Use path information to indicate where your Web application should look for information when servicing requests, or to divide you Web application into logical subservices.

## The request method type

Topic groups  See also

The *MethodType* property of an action item indicates what type of request messages it can process. The dispatcher compares the *MethodType* property of an action item to the *MethodType* of the request message. *MethodType* can take one of the following values:

| Value | Meaning |
|---|---|
| *mtGet* | The request is asking for the information associated with the target URI to be returned in a response message. |
| *mtHead* | The request is asking for the header properties of a response, as if servicing an *mtGet* request, but omitting the content of the response. |
| *mtPost* | The request is providing information to be posted to the Web application. |
| mtPut | The request asks that the resource associated with the target URI be replaced by the content of the request message. |
| *mtAny* | Matches any request method type, including *mtGet*, *mtHead*, *mtPut*, and *mtPost*. |

## Enabling and disabling action items

Topic groups  See also

Each action item has an *Enabled* property that can be used to enable or disable that action item. By setting *Enabled* to *False*, you disable the action item so that it is not considered by the dispatcher when it looks for an action item to handle a request.

A *BeforeDispatch* event handler can control which action items should process a request by changing the *Enabled* property of the action items before the dispatcher begins matching them to the request message.

**Caution:** Changing the *Enabled* property of an action during execution may cause unexpected results for subsequent requests. If the Web server application is a DLL that caches Web modules, the initial state will not be reinitialized for the next request. Use the *BeforeDispatch* event to ensure that all action items are correctly initialized to their appropriate starting states.

## Choosing a default action item

Topic groups  See also

Only one of the action items can be the default action item. The default action item is selected by setting its *Default* property to *True*. When the *Default* property of an action item is set to *True*, the *Default* property for the previous default action item (if any) is set to *False*.

When the dispatcher searches its list of action items to choose one to handle a request, it stores the name of the default action item. If the request has not been fully handled when the dispatcher reaches the end of its list of action items, it executes the default action item.

The dispatcher does not check the *PathInfo* or *MethodType* of the default action item. The dispatcher does not even check the *Enabled* property of the default action item. Thus, you can make sure the default action item is only called at the very end by setting its *Enabled* property to *False*.

The default action item should be prepared to handle any request that is encountered, even if it is only to return an error code indicating an invalid URI or *MethodType*. If the default action item does not handle the request, no response is sent to the Web client.

**Caution:** Changing the *Default* property of an action during execution may cause unexpected results for the current request. If the *Default* property of an action that has already been triggered is set to *True*, that action will not be re-evaluated and the dispatcher will not trigger that action when it reaches the end of the action list.

## Responding to request messages with action items

Topic groups See also

The real work of the Web server application is performed by action items when they execute. When the Web dispatcher fires an action item, that action item can respond to the current request message in two ways:

- If the action item has an associated producer component as the value of its *Producer* property, that producer automatically assigns the *Content* of the response message using its *Content* method. The Internet page of the component palette includes a number of content producer components that can help construct an HTML page for the content of the response message.
- After the producer has assigned any response content (if there is an associated producer), the action item receives an *OnAction* event. The *OnAction* event handler is passed the *TWebRequest* object that represents the HTTP request message and a *TWebResponse* object to fill with any response information.

If the action item's content can be generated by a single content producer, it is simplest to assign the content producer as the action item's *Producer* property. However, you can always access any content producer from the *OnAction* event handler as well. The *OnAction* event handler allows more flexibility, so that you can use multiple content producers, assign response message properties, and so on.

Both the content-producer component and the *OnAction* event handler can use any objects or runtime library methods to respond to request messages. They can access databases, perform calculations, construct or select HTML documents, and so on. For more information about generating response content using content-producer components, see Generating the content of response messages.

### Sending the response

An *OnAction* event handler can send the response back to the Web client by using the methods of the *TWebResponse* object. However, if no action item sends the response to the client, it will still get sent by the Web server application as long as the last action item to look at the request indicates that the request was handled.

### Using multiple action items

You can respond to a request from a single action item, or divide the work up among several action items. If the action item does not completely finish setting up the response message, it must signal this state in the *OnAction* event handler by setting the *Handled* parameter to *False*.

If many action items divide up the work of responding to request messages, each setting *Handled* to *False* so that others can continue, make sure the default action item leaves the *Handled* parameter set to *True*. Otherwise, no response will be sent to the Web client.

When dividing the work among several action items, either the *OnAction* event handler of the default action item or the *AfterDispatch* event handler of the dispatcher should check whether all the work was done and set an appropriate error code if it is not.

## Accessing client request information

Topic groups  See also

When an HTTP request message is received by the Web server application, the headers of the client request are loaded into the properties of a *TWebRequest* object. In NSAPI and ISAPI applications, the request message is encapsulated by a *TISAPIRequest* object. Console CGI applications use *TCGIRequest* objects, and Windows CGI applications use *TWinCGIRequest* objects.

The properties of the request object are read-only. You can use them to gather all of the information available in the client request, including

- Request header information
- The content of the request message

## Properties that contain request header information

Topic groups  See also

Most properties in a request object contain information about the request that comes from the HTTP request header. Not every request supplies a value for every one of these properties. Also, some requests may include header fields that are not surfaced in a property of the request object, especially as the HTTP standard continues to evolve. To obtain the value of a request header field that is not surfaced as one of the properties of the request object, use the *GetFieldByName* method.

The request header properties can be categorized by function:

- Properties that identify the target
- Properties that describe the Web client
- Properties that identify the purpose of the request
- Properties that describe the expected response
- Properties that describe the content

## Properties that identify the target

Topic groups  See also

The full target of the request message is given by the URL property. Usually, this is a URL that can be broken down into the protocol (HTTP), *Host* (server system), *ScriptName* (server application), *PathInfo* (location on the host), and *Query*.

Each of these pieces is surfaced in its own property. The protocol is always HTTP, and the *Host* and *ScriptName* identify the Web server application. The dispatcher uses the *PathInfo* portion when matching action items to request messages. The *Query* is used by some requests to specify the details of the requested information. Its value is also parsed for you as the *QueryFields* property.

## Properties that describe the Web client

Topic groups  See also

The request includes several properties that provide information about where the request originated. These include everything from the e-mail address of the sender (the *From* property), to the URI where the message originated (the *Referer* or *RemoteHost* property). If the request contains any content, and that content does not arise from the same URI as the request, the source of the content is given by the *DerivedFrom* property. You can also determine the IP address of the client (the *RemoteAddr* property), and the name and version of the application that sent the request (the *UserAgent* property).

## Properties that identify the purpose of the request

Topic groups  See also

The *Method* property is a string describing what the request message is asking the server application to do. The HTTP 1.1 standard defines the following methods:

| Value | What the message requests |
|---|---|
| OPTIONS | Information about available communication options. |
| GET | Information identified by the URL property. |
| HEAD | Header information from an equivalent GET message, without the content of the response. |
| POST | The server application to post the data included in the *Content* property, as appropriate. |
| PUT | The server application to replace the resource indicated by the URL property with the data included in the *Content* property. |
| DELETE | The server application to delete or hide the resource identified by the URL property. |
| TRACE | The server application to send a loop-back to confirm receipt of the request. |

The *Method* property may indicate any other method that the Web client requests of the server.

The Web server application does not need to provide a response for every possible value of *Method*. The HTTP standard does require that it service both GET and HEAD requests, however.

The *MethodType*property indicates whether the value of *Method* is GET (mtGet), HEAD (mtHead), POST (mtPost), PUT (mtPut) or some other string (mtAny). The dispatcher matches the value of the *MethodType* property with the *MethodType* of each action item.

## Properties that describe the expected response

Topic groups  See also

The *Accept* property indicates the media types the Web client will accept as the content of the response message. The *IfModifiedSince* property specifies whether the client only wants information that has changed recently. The *Cookie* property includes state information (usually added previously by your application) that can modify the response.

## Properties that describe the content

Topic groups  See also

Most requests do not include any content, as they are requests for information. However, some requests, such as POST requests, provide content that the Web server application is expected to use. The media type of the content is given in the *ContentType* property, and its length in the *ContentLength* property. If the content of the message was encoded (for example, for data compression), this information is in the *ContentEncoding* property. The name and version number of the application that produced the content is specified by the *ContentVersion* property. The *Title* property may also provide information about the content.

## The content of HTTP request messages

Topic groups  See also

In addition to the header fields, some request messages include a content portion that the Web server application should process in some way. For example, a POST request might include information that should be added to a database maintained by the Web server application.

The unprocessed value of the content is given by the *Content* property. If the content can be parsed into fields separated by ampersands (&), a parsed version is available in the *ContentFields* property.

## Creating HTTP response messages

Topic groups  See also

When the Web server application creates a *TWebRequest* object for an incoming HTTP request message, it also creates a corresponding *TWebResponse* object to represent the response message that will be sent in return. In NSAPI and ISAPI applications, the response message is encapsulated by a *TISAPIResponse* object. Console CGI applications use *TCGIResponse* objects, and Windows CGI applications use *TWinCGIResponse* objects.

The action items that generate the response to a Web client request fill in the properties of the response object. In some cases, this may be as simple as returning an error code or redirecting the request to another URI. In other cases, this may involve complicated calculations that require the action item to fetch information from other sources and assemble it into a finished form. Most request messages require some response, even if it is only the acknowledgment that a requested action was carried out.

Responding to HTTP requests involves

- Filling in the response header
- Setting the response content
- Sending the response

## Filling in the response header

Topic groups  See also

Most of the properties of the *TWebResponse* object represent the header information of the HTTP response message that is sent back to the Web client. An action item sets these properties from its *OnAction* event handler.

Not every response message needs to specify a value for every one of the header properties. The properties that should be set depend on the nature of the request and the status of the response.

Use the response object properties for

- Indicating the response status
- Indicating the need for client action
- Describing the server application
- Describing the content

## Indicating the response status

Topic groups  See also

Every response message must include a status code that indicates the status of the response. You can specify the status code by setting the *StatusCode* property. The HTTP standard defines a number of standard status codes with predefined meanings. In addition, you can define your own status codes using any of the unused possible values.

Each status code is a three-digit number where the most significant digit indicates the class of the response, as follows:

- 1xx: Informational (The request was received but has not been fully processed).
- 2xx: Success (The request was received, understood, and accepted).
- 3xx: Redirection (Further action by the client is needed to complete the request).
- 4xx: Client Error (The request cannot be understood or cannot be serviced).
- 5xx: Server Error (The request was valid but the server could not handle it).

Associated with each status code is a string that explains the meaning of the status code. This is given by the *ReasonString* property. For predefined status codes, you do not need to set the *ReasonString* property. If you define your own status codes, you should also set the *ReasonString* property.

## Indicating the need for client action

Topic groups  See also

When the status code is in the 300-399 range, the client must perform further action before the Web server application can complete its request. If you need to redirect the client to another URI, or indicate that a new URI was created to handle the request, set the *Location* property. If the client must provide a password before you can proceed, set the *WWWAuthenticate* property.

## Describing the server application

Topic groups  See also

Some of the response header properties describe the capabilities of the Web server application. The *Allow* property indicates the methods to which the application can respond. The *Server* property gives the name and version number of the application used to generate the response. The *Cookies* property can hold state information about the client's use of the server application which is included in subsequent request messages.

## Describing the content

Topic groups  See also

Several properties describe the content of the response. *ContentType* gives the media type of the response, and *ContentVersion* is the version number for that media type. *ContentLength* gives the length of the response. If the content is encoded (such as for data compression), indicate this with the *ContentEncoding* property. If the content came from another URI, this should be indicated in the *DerivedFrom* property. If the value of the content is time-sensitive, the *LastModified* property and the *Expires* property indicate whether the value is still valid. The *Title* property can provide descriptive information about the content.

## Setting the response content

Topic groups See also

For some requests, the response to the request message is entirely contained in the header properties of the response. In most cases, however, action item assigns some content to the response message. This content may be static information stored in a file, or information that was dynamically produced by the action item or its content producer.

You can set the content of the response message by using either the *Content* property or the *ContentStream* property.

The *Content* property is a string. Delphi strings are not limited to text values, so the value of the *Content* property can be a string of HTML commands, graphics content such as a bit-stream, or any other MIME content type.

Use the *ContentStream* property if the content for the response message can be read from a stream. For example, if the response message should send the contents of a file, use a *TFileStream* object for the *ContentStream* property. As with the *Content* property, *ContentStream* can provide a string of HTML commands or other MIME content type. If you use the *ContentStream* property, do not free the stream yourself. The Web response object automatically frees it for you.

**Note:** If the value of the *ContentStream* property is not **nil**, the *Content* property is ignored.

## Sending the response

Topic groups  See also

If you are sure there is no more work to be done in response to a request message, you can send a response directly from an *OnAction* event handler. The response object provides two methods for sending a response: *SendResponse* and *SendRedirect*. Call *SendResponse* to send the response using the specified content and all the header properties of the *TWebResponse* object. If you only need to redirect the Web client to another URI, the *SendRedirect* method is more efficient.

If none of the event handlers send the response, the Web application object sends it after the dispatcher finishes. However, if none of the action items indicate that they have handled the response, the application will close the connection to the Web client without sending any response.

## Generating the content of response messages

Topic groups See also

Delphi provides a number of objects to assist your action items in producing content for HTTP response messages. You can use these objects to generate strings of HTML commands that are saved in a file or sent directly back to the Web client. You can write your own content producers, deriving them from *TCustomContentProducer* or one of its descendants.

*TCustomContentProducer* provides a generic interface for creating any MIME type as the content of an HTTP response message. Its descendants include page producers and table producers:

- Page producers scan HTML documents for special tags that they replace with customized HTML code. They are described in Using page producer components.
- Table producers create HTML commands based on the information in a dataset. They are described in Using database information in responses.

## Using page producer components

Topic groups  See also

Page producers (*TPageProducer* and its descendants) take an HTML template and convert it by replacing special HTML-transparent tags with customized HTML code. You can store a set of standard response templates that are filled in by page producers when you need to generate the response to an HTTP request message. You can chain page producers together to iteratively build up an HTML document by successive refinement of the HTML-transparent tags.

## HTML templates

Topic groups  See also

An HTML template is a sequence of HTML commands and HTML-transparent tags. An HTML-transparent tag has the form

```
<#TagName Param1=Value1 Param2=Value2 ...>
```

The angle brackets (< and >) define the entire scope of the tag. A pound sign (#) immediately follows the opening angle bracket (<) with no spaces separating it from the angle bracket. The pound sign identifies the string to the page producer as an HTML-transparent tag. The tag name immediately follows the pound sign with no spaces separating it from the pound sign. The tag name can be any valid identifier and identifies the type of conversion the tag represents.

Following the tag name, the HTML-transparent tag can optionally include parameters that specify details of the conversion to be performed. Each parameter is of the form *ParamName*=*Value*, where there is no space between the parameter name, the equals symbol (=) and the value. The parameters are separated by whitespace.

The angle brackets (< and >) make the tag transparent to HTML browsers that do not recognize the #TagName construct.

When working with HTML templates, you will

- Optionally, Use predefined HTML-transparent tag Names
- Specify the HTML template
- Convert HTML-transparent tags

## Using predefined HTML-transparent tag names

Topic groups See also

While you can create your own HTML-transparent tags to represent any kind of information processed by your page producer, there are several predefined tag names associated with values of the *TTag* data type. These predefined tag names correspond to HTML commands that are likely to vary over response messages. They are listed in the following table:

| Tag Name | TTag value | What the tag should be converted to |
|---|---|---|
| *Link* | *tgLink* | A hypertext link. The result is an HTML sequence beginning with an <A> tag and ending with an </A> tag. |
| *Image* | *tgImage* | A graphic image. The result is an HTML <IMG> tag. |
| *Table* | *tgTable* | An HTML table. The result is an HTML sequence beginning with a <TABLE> tag and ending with a </TABLE> tag. |
| *ImageMap* | *tgImageMap* | A graphic image with associated hot zones. The result is an HTML sequence beginning with a <MAP> tag and ending with a </MAP> tag. |
| *Object* | *tgObject* | An embedded ActiveX object. The result is an HTML sequence beginning with an <OBJECT> tag and ending with an </OBJECT> tag. |
| *Embed* | *tgEmbed* | A Netscape-compliant add-in DLL. The result is an HTML sequence beginning with an <EMBED> tag and ending with an </EMBED> tag. |

Any other tag name is associated with *tgCustom*. The page producer supplies no built-in processing of the predefined tag names. They are simply provided to help applications organize the conversion process into many of the more common tasks.

**Note:** The predefined tag names are case insensitive.

## Specifying the HTML template

Topic groups  See also

Page producers provide you with many choices in how to specify the HTML template. You can set the *HTMLFile* property to the name of a file that contains the HTML template. You can set the *HTMLDoc* property to a *TStrings* object that contains the HTML template. If you use either the *HTMLFile* property or the *HTMLDoc* property to specify the template, you can generate the converted HTML commands by calling the *Content* method.

In addition, you can call the *ContentFromString* method to directly convert an HTML template that is a single string which is passed in as a parameter. You can also call the *ContentFromStream* method to read the HTML template from a stream. Thus, for example, you could store all your HTML templates in a memo field in a database, and use the *ContentFromStream* method to obtain the converted HTML commands, reading the template directly from a *TBlobStream* object.

## Converting HTML-transparent tags

Topic groups  See also

The page producer converts the HTML template when you call one of its *Content* methods. When the *Content* method encounters an HTML-transparent tag, it triggers the *OnHTMLTag* event. You must write an event handler to determine the type of tag encountered, and to replace it with customized content. See Using page producers from an action item for a simple example of converting HTML-transparent tags.

If you do not create an *OnHTMLTag* event handler for the page producer, HTML-transparent tags are replaced with an empty string.

## Using page producers from an action item

Topic groups  See also

A typical use of a page producer component uses the *HTMLFile* property to specify a file containing an HTML template. The *OnAction* event handler calls the *Content* method to convert the template into a final HTML sequence:

```
procedure WebModule1.MyActionEventHandler(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  PageProducer1.HTMLFile := 'Greeting.html';
  Response.Content := PageProducer1.Content;
end;
```

Greeting.html is a file that contains this HTML template:

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello <#UserName>!  Welcome to our web site.
</BODY>
</HTML>
```

The *OnHTMLTag* event handler replaces the custom tag (<#UserName>) in the HTML during execution:

```
procedureWebModule1.PageProducer1HTMLTag(Sender : TObject;Tag: TTag;
   const TagString: string; TagParams: TStrings;varReplaceText:string);
begin
  if CompareText(TagString,'UserName') = 0 then
    ReplaceText := TPageProducer(Sender).Dispatcher.Request.Content;
end;
```

If the content of the request message was the string *Mr. Ed*, the value of *Response.Content* would be

```
<HTML>
<HEAD><TITLE>Our brand new web site</TITLE></HEAD>
<BODY>
Hello Mr. Ed!  Welcome to our web site.
</BODY>
</HTML>
```

**Note:** This example uses an *OnAction* event handler to call the content producer and assign the content of the response message. You do not need to write an *OnAction* event handler if you assign the page producer's *HTMLFile* property at design time. In that case, you can simply assign *PageProducer1* as the value of the action item's *Producer* property to accomplish the same effect as the *OnAction* event handler above.

## Chaining page producers together

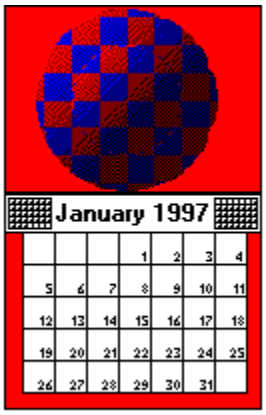
Topic groups  See also

The replacement text from an *OnHTMLTag* event handler need not be the final HTML sequence you want to use in the HTTP response message. You may want to use several page producers, where the output from one page producer is the input for the next.

The simplest way is to chain the page producers together is to associate each page producer with a separate action item, where all action items have the same *PathInfo* and *MethodType*. The first action item sets the content of the Web response message from its content producer, but its *OnAction* event handler makes sure the message is not considered handled. The next action item uses the *ContentFromString* method of its associated producer to manipulate the content of the Web response message, and so on. Action items after the first one use an *OnAction* event handler such as the following:

```
procedure WebModule1.Action2Action(Sender: TObject; Request: TWebRequest;
   Response: TWebResponse; var Handled: Boolean);
begin
  Response.Content := PageProducer2.ContentFromString(Response.Content);
end;
```

For example, consider an application that returns calendar pages in response to request messages that specify the month and year of the desired page. Each calendar page contains a picture, followed by the name and year of the month between small images of the previous month and next months, followed by the actual calendar. The resulting image looks something like this:




The general form of the calendar is stored in a template file. It looks like this:

```
<HTML>
<Head></HEAD>
<BODY>
<#MonthlyImage> <#TitleLine><#MainBody>
</BODY>
</HTML>
```

The *OnHTMLTag* event handler of the first page producer looks up the month and year from the request message. Using that information and the template file, it does the following:

- Replaces <#MonthlyImage> with <#Image Month=January Year=1997>.
- Replaces <#TitleLine> with <#Calendar Month=December Year=1996 Size=Small> January 1997 <#Calendar Month=February Year=1997 Size=Small>.
- Replaces <#MainBody> with <#Calendar Month=January Year=1997 Size=Large>.

The *OnHTMLTag* event handler of the next page producer uses the content produced by the first page producer, and replaces the <#Image Month=January Year=1997> tag with the appropriate HTML <IMG> tag. Yet another page producer resolves the #Calendar tags with appropriate HTML tables.

## Using database information in responses

Topic groups  See also

The response to an HTTP request message may include information taken from a database. Specialized content producers on the Internet palette page can generate the HTML to represent the records from a database in an HTML table.

To return database information in an HTTP response, you would typically

- Add a session to the Web module
- Represent the database information in HTML

As an alternate approach, special components on the Web MIDAS page of the component palette let you build Web servers that are part of a multi-tiered database application. See Building Web applications using InternetExpress for details.

## Adding a session to the Web module

Topic groups  See also

Both console CGI applications and Win-CGI applications are launched in response to HTTP request messages. When working with databases in these types of applications, you can use the default session to manage your database connections, because each request message has its own instance of the application. Each instance of the application has its own distinct, default session.

When writing an ISAPI application or an NSAPI application, however, each request message is handled in a separate thread of a single application instance. To prevent the database connections from different threads from interfering with each other, you must give each thread its own session.

Each request message in an ISAPI or NSAPI application spawns a new thread. The Web module for that thread is generated dynamically at runtime. Add a *TSession* object to the Web module to handle the database connections for the thread that contains the Web module.

Separate instances of the Web module are generated for each thread at runtime. Each of those modules contains the session object. Each of those sessions must have a separate name, so that the threads that handle separate request messages do not interfere with each other's database connections. To cause the session objects in each module to dynamically generate unique names for themselves, set the *AutoSessionName*property of the session object. Each session object will dynamically generate a unique name for itself and set the *SessionName* property of all datasets in the module to refer to that unique name. This allows all interaction with the database for each request thread to proceed without interfering with any of the other request messages. For more information on sessions, see Managing database sessions.

## Representing a dataset in HTML

Topic groups  See also

Specialized Content producer components on the Internet palette page supply HTML commands based on the records of a dataset. There are two types of data-aware content producers:

- The dataset page producer, which formats the fields of a dataset into the text of an HTML document.
- Table producers, which format the records of a dataset as an HTML table.

## Using dataset page producers

Topic groups  See also

Dataset page producers work like other page producer components: they convert a template that includes HTML-transparent tags into a final HTML representation. They include the special ability, however, of converting tags that have a tagname which matches the name of a field in a dataset into the current value of that field. For more information about using page producers in general, see Using page producer components.

To use a dataset page producer, add a *TDataSetPageProducer* component to your web module and set its *DataSet* property to the dataset whose field values should be displayed in the HTML content. Create an HTML template that describes the output of your dataset page producer. For every field value you want to display, include a tag of the form

```
<#FieldName>
```

in the HTML template, where *FieldName* specifies the name of the field in the dataset whose value should be displayed.

When your application calls the *Content*, *ContentFromString*, or *ContentFromStream* method, the dataset page producer substitutes the current field values for the tags that represent fields.

## Using table producers

Topic groups  See also

The Internet palette page includes two components that create an HTML table to represent the records of a dataset:

- Dataset table producers, which format the fields of a dataset into the text of an HTML document.
- Query table producers, which runs a query after setting parameters supplied by the request message and formats the resulting dataset as an HTML table.

Using either of the two table producers, you can customize the appearance of a resulting HTML table by specifying properties for the table's color, border, separator thickness, and so on. To set the properties of a table producer at design time, double-click the table producer component to display the Response Editor dialog.

- Specifying the table attributes
- Specifying the row attributes
- Specifying the columns
- Embedding tables in HTML documents

## Specifying the table attributes

Topic groups  See also

Table producers use the *THTMLTableAttributes* object to describe the visual appearance of the HTML table that displays the records from the dataset. The *THTMLTableAttributes* object includes properties for the table's width and spacing within the HTML document, and for its background color, border thickness, cell padding, and cell spacing. These properties are all turned into options on the HTML <TABLE> tag created by the table producer.

At design time, specify these properties using the Object Inspector. Select the table producer object in the Object Inspector and expand the *TableAttributes* property to access the display properties of the *THTMLTableAttributes* object.

You can also specify these properties programmatically at runtime.

## Specifying the row attributes

Topic groups See also

Similar to the table attributes, you can specify the alignment and background color of cells in the rows of the table that display data. The *RowAttributes* property is a *THTMLTableRowAttributes* object.

At design time, specify these properties using the Object Inspector by expanding the *RowAttributes* property. You can also specify these properties programmatically at runtime.

You can also adjust the number of rows shown in the HTML table by setting the *MaxRows* property.

## Specifying the columns

Topic groups  See also

If you know the dataset for the table at design time, you can use the Columns editor to customize the columns' field bindings and display attributes. Select the table producer component, and right-click. From the context menu, choose the Columns editor. This lets you add, delete, or rearrange the columns in the table. You can set the field bindings and display properties of individual columns in the Object Inspector after selecting them in the Columns editor.

If you are getting the name of the dataset from the HTTP request message, you can't bind the fields in the Columns editor at design time. However, you can still customize the columns programmatically at runtime, by setting up the appropriate *THTMLTableColumn* objects and using the methods of the *Columns* property to add them to the table. If you do not set up the *Columns* property, the table producer creates a default set of columns that match the fields of the dataset and specify no special display characteristics.

## Embedding tables in HTML documents

Topic groups  See also

You can embed the HTML table that represents your dataset in a larger document by using the *Header* and *Footer* properties of the table producer. Use *Header* to specify everything that comes before the table, and *Footer* to specify everything that comes after the table.

You may want to use another content producer (such as a page producer) to create the values for the *Header* and *Footer* properties.

If you embed your table in a larger document, you may want to add a caption to the table. Use the *Caption* and *CaptionAlignment* properties to give your table a caption.

## Using TDataSetTableProducer

Topic groups  See also

*TDataSetTableProducer* is a table producer that creates an HTML table for a dataset. Set the *DataSet* property of *TDataSetTableProducer* to specify the dataset that contains the records you want to display. You do not set the *DataSource* property, as you would for most data-aware objects in a conventional database application. This is because *TDataSetTableProducer* generates its own data source internally.

You can set the value of *DataSet* at design time if your Web application always displays records from the same dataset. You must set the *DataSet* property at runtime if you are basing the dataset on the information in the HTTP request message.

## Using TQueryTableProducer

Topic groups  See also

You can produce an HTML table to display the results of a query, where the parameters of the query come from the HTTP request message. Specify the *TQuery* object that uses those parameters as the *Query* property of a *TQueryTableProducer* component.

If the request message is a GET request, the parameters of the query come from the *Query* fields of the URL that was given as the target of the HTTP request message. If the request message is a POST request, the parameters of the query come from the content of the request message.

When you call the *Content* method of *TQueryTableProducer*, it runs the query, using the parameters it finds in the request object. It then formats an HTML table to display the records in the resulting dataset.

As with any table producer, you can customize the display properties or column bindings of the HTML table, or embed the table in a larger HTML document.

## Debugging server applications

Topic groups  See also

Debugging Web server applications presents some unique problems, because they run in response to messages from a Web server. You can not simply launch your application from the IDE, because that leaves the Web server out of the loop, and your application will not find the request message it is expecting. How you debug your Web server application depends on its type:

- Debugging ISAPI and NSAPI applications
- Debugging CGI and Win-CGI applications

## Debugging ISAPI and NSAPI applications

Topic groups  See also

ISAPI and NSAPI applications are actually DLLs that contain predefined entry points. The Web server passes request messages to the application by making calls to these entry points. You will need to set your application's run parameters to launch the server. Set up your breakpoints so that when the server passes a request message to your DLL, you hit one of your breakpoints, and can debug normally.

**Note:** Before launching the Web server using your application's run parameters, make sure that the server is not already running.

### Debugging under Windows NT

Under Windows NT, you must have the correct user rights to debug a DLL. In the User Manager, add your name to the lists granting rights for

- Log on as Service
- Act as part of the operation system
- Generate security audits

### Debugging with different Web servers

You must set the applications run parameters differently, depending on you Web server:

- Debugging with a Microsoft IIS server
- Debugging with a Windows 95 Personal Web Server
- Debugging with Netscape Server Version 2.0

## Debugging with a Microsoft IIS server

Topic groups  See also

To debug a Web server application using Microsoft IIS server (version 3 or earlier), choose Run|Parameters and set your application's run parameters as follows:

```
Host Application: c:\winnt\system32\inetsrv\inetinfo.exe
Run Parameters:   -e w3svc
```

This starts the IIS Server and allows you to debug your ISAPI DLL.

**Note:** Your directory path for *inetinfo.exe* may differ from the example.

If you are using version 4 or later, you must first make some changes to the Registry and the IIS Administration service:

1. Use DCOMCnfg to change the identity of the IIS Admin Service to your user account.
2. Use the Registry Editor (REGEDIT) or other utility to remove the "LocalService" keyword from all IISADMIN-related subkeys under HKEY_CLASSES_ROOT/AppID and HKEY_CLASSES_ROOT/CLSID. This keyword appears in the following subkeys:

   ```
   {61738644-F196-11D0-9953-00C04FD919C1} // IIS WAMREG admin Service
   {9F0BD3A0-EC01-11D0-A6A0-00A0C922E752} // IIS Admin Crypto Extension
   {A9E69610-B80D-11D0-B9B9-00A0C922E750} // IISADMIN Service
   ```

   In addition, under the AppID node, remove the "RunAs" keyword from the first two of these subkeys. To the last subkey listed, add "Interactive User" as the value of the "RunAs" keyword.
3. Still using the Registry Editor, add "LocalService32" subkeys to all IISADMIN-related subkeys under the CLSID node. That is, for every subkey listed in step 2 (and any others under which you found the "LocalService" keyword), add a "LocalService32" subkey under the CLSID/<subkey> node. Set the default value of these new keys to

   ```
   c:\\winnt\\system32\\inetsrv\\inetinfo.exe -e w3svc
   ```

   (the path for inetinfo.exe may differ for your system).
4. Add a value of "dword:3" to the "Start" keyword for the following subkeys:

   ```
   [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\IISADMIN]
   [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\MSDTC]
   [HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\W3SVC]
   ```
5. Stop the WWW, FTP, and IISAdmin services from the Microsoft Management Console or the Services dialog box in the control panel. As an alternative, you can simply do KILL INETINFO using KILL.EXE from the NT Resource Kit.

Now you are ready to debug in the same way as when using IIS version 3 or earlier. That is, choose Run|Parameters and set your application's run parameters as follows:

```
Host Application: c:\winnt\system32\inetsrv\inetinfo.exe
Run Parameters:   -e w3svc
```

**Note:** When you have finished debugging, you will need to back out all the Registry changes you made in steps 2 through 4.

### Debugging under MTS

Another approach you can take when using IIS is to configure your Web directory as an MTS Library package. You can then debug your ISAPI dll by running it under MTS.

To configure the Web directory as an MTS Library package, use the following steps:

1. Start the Internet Service Manager. You should see both the Internet Information Server and the Microsoft Transaction Server trees.
2. Expand the Internet Information Server tree to view the items under "Default Web Site". Select the Web directory where your ISAPI dll is installed. Right click and choose Properties.
3. On the Virtual Directory tab page, check Run in separate memory space (isolated process), and click OK.
4. Expand the Microsoft Transaction Server tree to view the items under "Packages Installed". Right click on the "Packages Installed" node and select Refresh.
5. You will see a package with the same suffix as the Web directory. Right click this package and

choose Properties.

6 On the Identity tab page, select the Interactive User radio button, and click OK.

The previous steps configure your Web directory. After doing so, you can debug your ISAPI DLL as follows:

1 In Delphi, choose Run|Parameters. In the Host Application field, enter the fully qualified pathname of the MTS executable. Typically, this is

```
c:\winnt\system32\mtx.exe
```

2 In the parameters field, you must use the /p option with the name of the MTS package. To get this value, start the Internet Service Manager and expand the Microsoft Transaction Server tree to view the items under “Packages Installed”. Right click on the package with the same suffix as the Web directory. Choose Properties, and copy the package name from the General tab page to the clipboard.

In the parameters field, paste the name of the package, and then surround it with double quotes and precede the quoted string with /p:. The resulting parameters field should look similar to the following:

```
/p:"IIS-{Default Web Site//ROOT/WEBPUB/DEMO}"
```

Note that there should not be a space between the colon and the package name.

**Tip:** When the Web directory is installed as an MTS package, you can also use MTS to easily shut down the DLL. Just expand the Microsoft Transaction Server tree in the Internet Service Manager so that you can see the items under the “Packages Installed” node. Right click on the package with the same suffix as the Web directory and choose Shut Down.

## Debugging with a Windows 95 Personal Web Server

Topic groups  See also

To debug a Web server application using Personal Web Server, set your application's run parameters as follows:

```
Host Application: c:\Program Files\websvc\system\inetsw95.exe
Run Parameters:   -w3svc
```

This starts the Personal Web Server and allows you to debug your ISAPI DLL.

**Note:** Your directory path for *inetsw95.exe* may differ from the example.

## Debugging with Netscape Server Version 2.0

Topic groups  See also

Before using Web server applications on Netscape servers, you must make certain configuration changes.

First, copy the ISAPITER.DLL file (from the Bin directory) into the C:\Netscape\Server\Nsapi\Examples directory. (Your directory path may differ.)

Next, make the following modifications to the server configuration files located in the C:\Netscape\Server\Httpd-<servername>\Config directory.

1. In the OBJ.CONF file, insert the line

   ```
   Init funcs="handle_isapi,check_isapi,log_isapi" fn="load_modules"
     shlib="c:/netscape/server/nsapi/examples/ISAPIter.dll"
   ```

   after the line

   ```
   Init fn=load-types mime-types=mime.types
   ```

2. In the <Object name=default> section of OBJ.CONF, insert the lines

   ```
   NameTrans from="/scripts" fn="pfx2dir" dir="C:/Netscape/Server/docs/scripts"
     name="isapi"
   ```

   before the line

   ```
   NameTrans fn=document-root root="C:/Netscape/Server/docs"
   ```

3. Add the following section to the end of OBJ.CONF:

   ```
   <Object name="isapi">
   PathCheck fn="check_isapi"
   ObjectType fn="force-type" type="magnus-internal/isapi"
   Service fn="handle_isapi"
   </Object>
   ```

4. Add the following line to the end of the MIME.TYPES file:

   ```
   type=magnus-internal/isapi exts=dll
   ```

   This should be the last line in the file.

**Note:** Line breaks are included in steps 1 and 2 above only to enhance readability. Do not type carriage-returns when you place these lines in configuration files.

To debug a Web server application using Netscape Fast Track server, set the application's run parameters as follows:

```
Host Application: c:\Netscape\server\bin\httpd\httpd.exe
Run Parameters:   c:\Netscape\server\httpd-<servername>\config
```

This starts the server and indicates to the server where the configuration files are located.

## Debugging CGI and Win-CGI applications

Topic groups See also

It is more difficult to debug CGI and Win-CGI applications, because the application itself must be launched by the Web server.

### Simulating the server

For Win-CGI applications, you can simulate the server by manually writing the configuration settings file that contains the request information. Then launch the Win-CGI application, passing the location of the file containing the client information and the location of a file that the Win-CGI program should use to create content. You can then debug normally.

### Debugging as a DLL

Another approach you can take with both CGI and Win-CGI applications is first to create and debug your application as an ISAPI or NSAPI application. Once your ISAPI or NSAPI application is working smoothly, convert it to a CGI or Win-CGI application. To convert your application, use the following steps:

1.Right-click the Web module and choose Add To Repository.

2. In the Add To Repository dialog, give your Web module a title, text description, repository page (probably Data Modules), author name, and icon.

8.Choose OK to save your web module as a template.

4.From the main menu, choose File|New and select Web Server Application. In the New Web Server Application dialog, choose CGI or Win-CGI, as appropriate.

5.Delete the automatically generated Web Module.

6.From the main menu, choose File | New and select the template you saved in step 3. This will be on the page you specified in step 2.

CGI and Win-CGI applications are simpler than ISAPI and NSAPI applications. Each instance of a CGI or Win-CGI application must handle only a single thread. Thus, these applications do not encounter the multi-threading issues that ISAPI and NSAPI applications must deal with. They also are immune to the problems that can arise from the caching of Web modules in ISAPI and NSAPI applications.

## Working with sockets

Topic groups  See also

The socket components let you create an application that can communicate with other systems using TCP/IP and related protocols. Using sockets, you can read and write over connections to other machines without worrying about the details of the actual networking software. Sockets provide connections based on the TCP/IP protocol, but are sufficiently general to work with related protocols such as Xerox Network System (XNS), Digital's DECnet, or Novell's IPX/SPX family.

Using sockets, you can write network servers or client applications that read from and write to other systems. A server or client application is usually dedicated to a single service such as Hypertext Transfer Protocol (HTTP) or File Transfer Protocol (FTP). Using server sockets, an application that provides one of these services can link to client applications that want to use that service. Client sockets allow an application that uses one of these services to link to server applications that provide the service.

To write applications that use sockets, you should understand

- Implementing services
- Types of socket connections
- Describing sockets
- Using socket components
- Responding to socket events
- Reading and writing over socket connections

## Implementing services

Topic groups  See also

Sockets provide one of the pieces you need to write network servers or client applications. For many services, such as HTTP or FTP, third party servers are readily available. Some are even bundled with the operating system, so that there is no need to write one yourself. However, when you want more control over the way the service is implemented, a tighter integration between your application and the network communication, or when no server is available for the particular service you need, then you may want to create your own server or client application. For example, when working with distributed data sets, you may want to write a layer to communicate with databases on other systems.

To implement or use a service using sockets, you must understand

- service protocols
- services and ports

## Service protocols

Topic groups See also

Before you can write a network server or client, you must understand the service that your application is providing or using. Many services have standard protocols that your network application must support. If you are writing a network application for a standard service such as HTTP, FTP, or even finger or time, you must first understand the protocols used to communicate with other systems. See the documentation on the particular service you are providing or using.

If you are providing a new service for an application that communicates with other systems, the first step is designing the communication protocol for the servers and clients of this service. What messages are sent? How are these messages coordinated? How is the information encoded?

### Communicating with applications

Often, your network server or client application provides a layer between the networking software and an application that uses the service. For example, an HTTP server sits between the Internet and a Web server application that provides content and responds to HTTP request messages.

Sockets provide the interface between your network server or client application and the networking software. You must provide the interface between your application and the applications that use it. You can copy the API of a standard third party server (such as ISAPI), or you can design and publish your own API.

## Services and ports

Topic groups  See also

Most standard services are associated, by convention, with specific port numbers. When implementing services, you can consider the port number a numeric code for the service.

If you are implementing a standard service, Windows socket objects provide methods for you to look up the port number for the service. If you are providing a new service, you can specify the associated port number in a SERVICES file on Windows 95 or NT machines. See the Microsoft documentation for Windows sockets for more information on setting up a SERVICES file.

## Types of socket connections

Topic groups  See also

Socket connections can be divided into three basic types, which reflect how the connection was initiated and what the local socket is connected to. These are

- Client connections.
- Listening connections.
- Server connections.

Once the connection to a client socket is completed, the server connection is indistinguishable from a client connection. Both end points have the same capabilities and receive the same types of events. Only the listening connection is fundamentally different, as it has only a single endpoint.

## Client connections

Topic groups  See also

Client connections connect a client socket on the local system to a server socket on a remote system. Client connections are initiated by the client socket. First, the client socket must describe the server socket it wishes to connect to. The client socket then looks up the server socket and, when it locates the server, requests a connection. The server socket may not complete the connection right away. Server sockets maintain a queue of client requests, and complete connections as they find time. When the server socket accepts the client connection, it sends the client socket a full description of the server socket to which it is connecting, and the connection is completed by the client.

## Listening connections

Topic groups  See also

Server sockets do not locate clients. Instead, they form passive “half connections” that listen for client requests. Server sockets associate a queue with their listening connections; the queue records client connection requests as they come in. When the server socket accepts a client connection request, it forms a new socket to connect to the client, so that the listening connection can remain open to accept other client requests.

## Server connections

Topic groups  See also

Server connections are formed by server sockets when a listening socket accepts a client request. A description of the server socket that completes the connection to the client is sent to the client when the server accepts the connection. The connection is established when the client socket receives this description and completes the connection.

## Describing sockets

Topic groups  See also

Sockets let your network application communicate with other systems over the network. Each socket can be viewed as an endpoint in a network connection. It has an address that specifies

- The system it is running on.
- The types of interfaces it understands.
- The port it is using for the connection.

A full description of a socket connection includes the addresses of the sockets on both ends of the connection. You can describe the address of each socket endpoint by supplying both the IP address or host and the port number.

Before you can make a socket connection, you must fully describe the sockets that form its end points. Some of the information is available from the system your application is running on. For instance, you do not need to describe the local IP address of a client socket—this information is available from the operating system.

The information you must provide depends on the type of socket you are working with. Client sockets must describe the server they want to connect to. Listening server sockets must describe the port that represents the service they provide.

## Describing the host

Topic groups  See also

The host is the system that is running the application that contains the socket. You can describe the host for a socket by giving its IP address, which is a string of four numeric (byte) values in the standard Internet dot notation, such as

```
123.197.1.2
```

A single system may support more than one IP address.

IP addresses are often difficult to remember and easy to mistype. An alternative is to use the host name. Host names are aliases for the IP address that you often see in Uniform Resource Locators (URLs). They are strings containing a domain name and service, such as

```
http://www.wSite.Com
```

Most Intranets provide host names for the IP addresses of systems on the Internet. On Windows 95 and NT machines, if a host name is not available, you can create one for your local IP address by entering the name into the HOSTS file. See the Microsoft documentation on Windows sockets for more information on the HOSTS file.

Server sockets do not need to specify a host. The local IP address can be read from the system. If the local system supports more than one IP address, server sockets will listen for client requests on all IP addresses simultaneously. When a server socket accepts a connection, the client socket provides the remote IP address.

Client sockets must specify the remote host by providing either its host name or IP address.

### Choosing between a host name and an IP address

Most applications use the host name to specify a system. Host names are easier to remember, and easier to check for typographical errors. Further, servers can change the system or IP address that is associated with a particular host name. Using a host name allows the client socket to find the abstract site represented by the host name, even when it has moved to a new IP address.

If the host name is unknown, the client socket must specify the server system using its IP address. Specifying the server system by giving the IP address is faster. When you provide the host name, the socket must search for the IP address associated with the host name, before it can locate the server system.

## Using ports

Topic groups  See also

While the IP address provides enough information to find the system on the other end of a socket connection, you also need a port number on that system. Without port numbers, a system could only form a single connection at a time. Port numbers are unique identifiers that enable a single system to host multiple connections simultaneously, by giving each connection a separate port number.

One way to look at port numbers is as numeric codes for the services implemented by network applications. This is a convention that allows listening server connections to make themselves available on a fixed port number so that they can be found by client sockets. Server sockets listen on the port number associated with the service they provide. When they accept a connection to a client socket, they create a separate socket connection that uses a different, arbitrary, port number. This way, the listening connection can continue to listen on the port number associated with the service.

Client sockets use an arbitrary local port number, as there is no need for them to be found by other sockets. They specify the port number of the server socket to which they want to connect so that they can find the server application. Often, this port number is specified indirectly, by naming the desired service.

## Using socket components

Topic groups  See also

The Internet palette page includes two socket components (client sockets and server sockets) that allow your network application to form connections to other machines, and that allow you to read and write information over that connection. Associated with each of these socket components are Windows socket objects, which represent the endpoint of an actual socket connection. The socket components use the Windows socket objects to encapsulate the Windows socket API calls, so that your application does not need to be concerned with the details of establishing the connection or managing the socket messages.

If you want to work with the Windows socket API calls, or customize the details of the connections that the socket components make on your behalf, you can use the properties, events, and methods of the Windows socket objects.

## Using client sockets

Topic groups  See also

Add a client socket component (*TClientSocket*) to your form or data module to turn your application into a TCP/IP client. Client sockets allow you to specify the server socket you want to connect to, and the service you want that server to provide. Once you have described the desired connection, you can use the client socket component to complete the connection to the server.

Each client socket component uses a single client Windows socket object (*TClientWinSocket*) to represent the client endpoint in a connection.

Use client sockets to

- Specify the desired server.
- Connect to the server.
- Get information about the connection.
- Read from or write to the server.
- Close the connection.

## Specifying the desired server

Topic groups  See also

Client socket components have a number of properties that allow you to specify the server system and port to which you want to connect. You can specify the server system by its host name using the *Host* property. If you do not know the host name, or if you are concerned about the speed of locating the server, you can specify the IP address of the server system by using the *Address* property. You must specify either a host name or an IP address. If you specify both, the client socket component will use the host name.

In addition to the server system, you must specify the port on the server system that your client socket will connect to. You can specify the server port number directly using the *Port* property, or indirectly by naming the desired service using the *Service* property. If you specify both the port number and the service, the client socket component will use the service name.

## Forming the connection

Topic groups  See also

Once you have set the properties of your client socket component to describe the server you want to connect to, you can form the connection at runtime by calling the *Open* method. If you want your application to form the connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

## Getting information about the connection

Topic groups  See also

After completing the connection to a server socket, you can use the client Windows socket object associated with your client socket component to obtain information about the connection. Use the *Socket* property to get access to the client Windows socket object. This Windows socket object has properties that enable you to determine the address and port number used by the client and server sockets to form the end points of the connection. You can use the *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. You can use the *Handle* property to access the window that receives messages from the socket connection. The *ASyncStyles* property determines what types of messages that window handle receives.

## Closing the connection

Topic groups  See also

When you have finished communicating with a server application over the socket connection, you can shut down the connection by calling the *Close* method. The connection may also be closed from the server end. If that is the case, you will receive notification in an *OnDisconnect* event.

## Using server sockets

Topic groups  See also

Add a server socket component (*TServerSocket*) to your form or data module to turn your application into a TCP/IP server. Server sockets allow you to specify the service you are providing or the port you want to use to listen for client requests. You can use the server socket component to listen for and accept client connection requests.

Each server socket component uses a single server Windows socket object (*TServerWinSocket*) to represent the server endpoint in a listening connection. It also uses a server client Windows socket object (*TServerClientWinSocket*) for the server endpoint of each active connection to a client socket that the server accepts.

Use server sockets to

- Specify the port.
- Listen for client requests.
- Connect to clients.
- Get information about client connections.
- Read from or write to the server.
- Close server connections.

## Specifying the desired server

Topic groups  See also

Before your server socket can listen to client requests, you must specify the port that your server will listen on. You can specify this port using the *Port* property. If your server application is providing a standard service that is associated by convention with a specific port number, you can specify the port number indirectly using the *Service* property. It is a good idea to use the *Service* property, as it is easy to miss typographical errors made when setting the port number. If you specify both the *Port* property and the *Service* property, the server socket will use the service name.

## Listening for client requests

Topic groups  See also

Once you have set the port number of your server socket component, you can form a listening connection at runtime by calling the *Open* method. If you want your application to form the listening connection automatically when it starts up, set the *Active* property to *True* at design time, using the Object Inspector.

## Connecting to clients

Topic groups  See also

A listening server socket component automatically accepts client connection requests when they are received. You receive notification every time this occurs in an *OnClientConnect* event.

## Getting information about connections

Topic groups  See also

Once you have opened a listening connection with your server socket, you can use the server Windows socket object associated with your server socket component to obtain information about the connection. Use the *Socket* property to get access to the server Windows socket object. This Windows socket object has properties that enable you to find out about all the active connections to client sockets that were accepted by your server socket component. Use the *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. Use the *Handle* property to access the window that receives messages from the socket connection.

Each active connection to a client application is encapsulated by a server client Windows socket object (*TServerClientWinSocket*). You can access all of these through the *Connections* property of the server Windows socket object. These server client Windows socket objects have properties that enable you to determine the address and port number used by the client and server sockets which form the end points of the connection. You can use the *SocketHandle* property to obtain a handle to the socket connection to use when making Windows socket API calls. You can use the *Handle* property to access the window that receives messages from the socket connection. The *ASyncStyles* property determines what types of messages that window handle receives.

## Closing server connections

Topic groups  See also

When you want to shut down the listening connection, call the *Close* method. This shuts down all open connections to client applications, cancels any pending connections that have not been accepted, and then shuts down the listening connection so that your server socket component does not accept any new connections.

When clients shut down their individual connections to your server socket, you are informed by an *OnClientDisconnect* event.

## Responding to socket events

Topic groups  See also

When writing applications that use sockets, most of the work usually takes place in event handlers of the socket components. Some sockets generate reading and writing events when it is time to begin reading or writing over the socket connection.

Client sockets receive an *OnDisconnect* event when the server ends a connection, and server sockets receive an *OnClientDisconnect* event when the client ends a connection.

Both client sockets and server sockets generate error events when they receive error messages from the connection.

Socket components also receive a number of events in the course of opening and completing a connection. If your application needs to influence how the opening of the socket proceeds, or if it should start reading or writing once the connection is formed, you will want to write event handlers to respond to these client events or server events.

## Error events

Topic groups  See also

Client sockets generate an *OnError* event when they receive error messages from the connection. Server sockets generate an *OnClientError.* You can write an *OnError* or *OnClientError* event handler to respond to these error messages. The event handler is passed information about

- What Windows socket object received the error notification.
- What the socket was trying to do when the error occurred.
- The error code that was provided by the error message.

You can respond to the error in the event handler, and change the error code to 0 to prevent the socket from raising an exception.

## Client events

Topic groups  See also

When a client socket opens a connection, the following events occur:

1. An *OnLookup* event occurs prior to an attempt to locate the server socket. At this point you can not change the *Host*, *Address*, *Port,* or *Service* properties to change the server socket that is located. You can use the *Socket* property to access the client Windows socket object, and use its *SocketHandle* property to make Windows API calls that affect the client properties of the socket. For example, if you want to set the port number on the client application, you would do that now before the server client is contacted.
2. The Windows socket is set up and initialized for event notification.
3. An *OnConnecting* event occurs after the server socket is located. At this point, the Windows Socket object available through the *Socket* property can provide information about the server socket that will form the other end of the connection. This is the first chance to obtain the actual port and IP address used for the connection, which may differ from the port and IP address of the listening socket that accepted the connection.
4. The connection request is accepted by the server and completed by the client socket.
5. An *OnConnect* event occurs after the connection is established. If your socket should immediately start reading or writing over the connection, write an *OnConnect* event handler to do it.

## Server events

Topic groups  See also

Server socket components form two types of connections: listening connections and connections to client applications. The server socket receives events during the formation of each of these connections.

### Events when listening

Just before the listening connection is formed, the *OnListen* event occurs. At this point you can obtain the server Windows socket object through the *Socket* property. You can use its *SocketHandle* property to make changes to the socket before it is opened for listing. For example, if you want to restrict the IP addresses the server uses for listening, you would do that in an *OnListen* event handler.

### Events with client connections

When a server socket accepts a client connection request, the following events occur:

1. The server socket generates an *OnGetSocket* event, passing in the Windows socket handle for the socket that forms the server endpoint of the connection. If you want to provide your own customized descendant of *TServerClientWinSocket*, you can create one in an *OnGetSocket* event handler, and that will be used instead of *TServerClientWinSocket*.
2. An *OnAccept* event occurs, passing in the new *TServerClientWinSocket* object to the event handler. This is the first point when you can use the properties of *TServerClientWinSocket* to obtain information about the server endpoint of the connection to a client.
3. If *ServerType* is *stThreadBlocking* an *OnGetThread* event occurs. If you want to provide your own customized descendant of *TServerClientThread*, you can create one in an *OnGetThread* event handler, and that will be used instead of *TServerClientThread*. For more information on creating custom server client threads, see Writing server threads.
4. If *ServerType* is *stThreadBlocking*, an *OnThreadStart* event occurs as the thread begins execution. If you want to perform any initialization of the thread, or make any Windows socket API calls before the thread starts reading or writing over the connection, use the *OnThreadStart* event handler.
5. The client completes the connection and an *OnClientConnect* event occurs. With a non-blocking server, you may want to start reading or writing over the socket connection at this point.

## Reading and writing over socket connections

Topic groups  See also

The reason you form socket connections to other machines is so that you can read or write information over those connections. What information you read or write, or when you read it or write it, depends on the service associated with the socket connection.

Reading and writing over sockets can occur asynchronously, so that it does not block the execution of other code in your network application. This is called a non-blocking connection. You can also form blocking connections, where your application waits for the reading or writing to be completed before executing the next line of code.

## Non-blocking connections

Topic groups  See also

Non-blocking connections read and write asynchronously, so that the transfer of data does not block the execution of other code in you network application. To create a non-blocking connection

- On client sockets, set the *ClientType* property to *ctNonBlocking*.
- On server sockets, set the *ServerType* property to *stNonBlocking*.

When the connection is non-blocking, reading and writing events inform your socket when the socket on the other end of the connection tries to read or write information.

## Reading and writing events

Topic groups  See also

Non-blocking sockets generate reading and writing events that inform your socket when it needs to read or write over the connection. With client sockets, you can respond to these notifications in an *OnRead* or *OnWrite* event handler. With server sockets, you can respond to these events in an *OnClientRead* or *OnClientWrite* event handler.

The Windows socket object associated with the socket connection is provided as a parameter to the read or write event handlers. This Windows socket object provides a number of methods to allow you to read or write over the connection.

To read from the socket connection, use the *ReceiveBuf* or *ReceiveText* method. Before using the *ReceiveBuf* method, use the *ReceiveLength* method to get an estimate of the number of bytes the socket on the other end of the connection is ready to send.

To write to the socket connection, use the *SendBuf*, *SendStream*, or *SendText* method. If you have no more need of the socket connection after you have written your information over the socket, you can use the *SendStreamThenDrop* method. *SendStreamThenDrop* closes the socket connection after writing all information that can be read from the stream. If you use the *SendStream* or *SendStreamThenDrop* method, do not free the stream object. The socket frees the stream automatically when the connection is closed.

**Note:** *SendStreamThenDrop* will close down a server connection to an individual client, not a listening connection.

## Blocking connections

Topic groups  See also

When the connection is blocking your socket must initiate reading or writing over the connection rather than waiting passively for a notification from the socket connection. Use a blocking socket when your end of the connection is in charge of when reading and writing takes place.

For client sockets, set the *ClientType* property to *ctBlocking* to form a blocking connection. Depending on what else your client application does, you may want to create a new execution thread for reading or writing, so that your application can continue executing code on other threads while it waits for the reading or writing over the connection to be completed.

For server sockets, set the *ServerType* property to *stThreadBlocking* to form a blocking connection. Because blocking connections hold up the execution of all other code while the socket waits for information to be written or read over the connection, server socket components always spawn a new execution thread for every client connection when the *ServerType* is *stThreadBlocking*.

Most applications that use blocking connections are written using threads.

Even if you do not use threads, you will want to read and write using TWinSocketStream.

## Using threads with blocking connections

Topic groups  See also

Client sockets do not automatically spawn new threads when reading or writing using a blocking connection. If your client application has nothing else to do until the information has been read or written, this is what you want. If your application includes a user interface that must still respond to the user, however, you will want to generate a separate thread for the reading or writing.

When server sockets form blocking connections, they always spawn separate threads for every client connection, so that no client must wait until another client has finished reading or writing over the connection. By default, server sockets use *TServerClientThread* objects to implement the execution thread for each connection.

*TServerClientThread* objects simulate the *OnClientRead* and *OnClientWrite* events that occur with non-blocking connections. However, these events occur on the listening socket, which is not thread-local. If client requests are frequent, you will want to create your own descendant of *TServerClientThread* to provide thread-safe reading and writing.

When writing client threads or writing server threads, you can use TWinSocketStream to do the actual reading and writing.

## Using TWinSocketStream

Topic groups  See also

When implementing the thread for a blocking connection, you must determine when the socket on the other end of the connection is ready for reading or writing. Blocking connections do not notify the socket when it is time to read or write. To see if the connection is ready, use a *TWinSocketStream* object. *TWinSocketStream* provides methods to help coordinate the timing of reading and writing. Call the *WaitForData* method to wait until the socket on the other end is ready to write.

When reading or writing using *TWinSocketStream*, the stream times out if the reading or writing has not completed after a specified period of time. As a result of this timing out, the socket application won't hang endlessly trying to read or write over a dropped connection.

**Note:** You can not use *TWinSocketStream* with a non-blocking connection.

## Writing client threads

Topic groups  See also

To write a thread for client connections, define a new thread object using the New Thread Object dialog. The *Execute* method of your new thread object handles the details of reading and writing over the thread connection. It creates a *TWinSocketStream* object, and uses that to read or write. For example:

```
procedure TMyClientThread.Execute;
var
  TheStream: TWinSocketStream;
  buffer: string;
begin
  { create a TWinSocketStream for reading and writing }
  TheStream := TWinSocketStream.Create(ClientSocket1.Socket, 60000);
  try
    { fetch and process commands until the connection or thread is terminated }
    while (not Terminated) and (ClientSocket1.Active) do
    begin
      try
        GetNextRequest(buffer); { GetNextRequest must be a thread-safe method }
        { write the request to the server }
        TheStream.Write(buffer, Length(buffer) + 1);
        { continue the communication (e.g. read a response from the server) }
        ...
      except
        if not(ExceptObject is EAbort) then
          Synchronize(HandleThreadException); { you must write HandleThreadException
}
      end;
    end;
  finally
   TheStream.free;
  end;
end;
```

To use your thread, create it in an *OnConnect* event handler. For more information about creating and running threads, see Executing thread objects.

## Writing server threads

Topic groups  See also

Threads for server connections are descendants of *TServerClientThread*. Because of this, you can't use the New Thread object dialog. Instead, declare your thread manually as follows:

```
TMyServerThread = class(TServerClientThread);
```

To implement this thread, you override the *ClientExecute* method instead of the *Execute* method.

Implementing the *ClientExecute* method is much the same as writing the *Execute* method of the thread for a client connection. However, instead of using a client socket component that you place in your application from the Component palette, the server client thread must use the *TServerClientWinSocket* object that is created when the listening server socket accepts a client connection. This is available as the public *ClientSocket* property. In addition, you can use the protected *HandleException* method rather than writing your own thread-safe exception handling. For example:

```
procedure TMyServerThread.ClientExecute;
var
  Stream : TWinSocketStream;
  Buffer : array[0 .. 9] of Char;
begin
  { make sure connection is active }
  while (not Terminated) and ClientSocket.Connected do
  begin
    try
      Stream := TWinSocketStream.Create(ClientSocket, 60000);
      try
        FillChar(Buffer, 10, 0); { initialize the buffer }
        { give the client 60 seconds to start writing }
        if Stream.WaitForData(60000) then
        begin
          if Stream.Read(Buffer, 10) = 0 then { if can't read in 60 seconds }
            ClientSocket.Close;               { close the connection }
          { now process the request }
          ...
        end
        else
          ClientSocket.Close; { if client doesn't start, close }
      finally
        Stream.Free;
      end;
    except
      HandleException;
    end;
  end;
end;
```

**Warning:** Server sockets cache the threads they use. Be sure the *ClientExecute* method performs any necessary initialization so that there are no adverse results from changes made when the thread last executed.

To use your thread, create it in an *OnGetThread* event handler. When creating the thread, set the *CreateSuspended* parameter to *False*.

## Writing CORBA applications

Topic groups  See also

Delphi provides wizards and classes to make it easy to create distributed applications based on the Common Object Request Broker Architecture (CORBA). CORBA is a specification adopted by the Object Management Group (OMG) to address the complexity of developing distributed object applications.

As its name implies, CORBA provides an object-oriented approach to writing distributed applications. This is in contrast to a message-oriented approach such as the one described for HTTP applications in Creating Internet server applications. Under CORBA, server applications implement objects that can be used remotely by client applications, through well-defined interfaces.

**Note:** COM provides another object-oriented approach to distributed applications. For more information about COM, see Overview of COM technologies. Unlike COM, however, CORBA is a standard that applies to platforms other than Windows. This means you can write CORBA clients or servers using Delphi that can communicate with CORBA-enabled applications running on other platforms.

The CORBA specification defines how client applications communicate with objects that are implemented on a server. This communication is handled by an Object Request Broker (ORB). Delphi's CORBA support is based on the VisiBroker for C++ ORB (Version 3.3.2) with a special wrapper (orbpas.dll) that exposes a subset of the ORB functionality to Delphi applications.

In addition to the basic ORB technology, which enables clients to communicate with objects on server machines, the CORBA standard defines a number of standard services. Because these services use well-defined interfaces, developers can write clients that use these services even if the servers are written by different vendors.

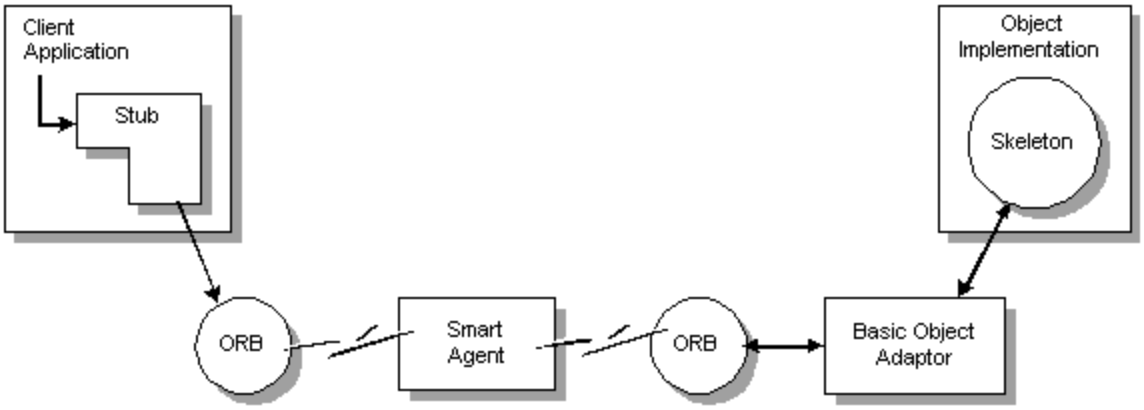
For more information on developing CORBA applications in Delphi, see:

- Overview of a CORBA application
- Writing CORBA servers
- Writing CORBA clients
- Customizing CORBA applications
- Deploying CORBA applications

## Overview of a CORBA application

Topic groups  See also

If you are already doing object-oriented programming, CORBA makes writing distributed applications easy, because it lets you use remote objects almost as if they were local. This is because the design of a CORBA application is much like any other object-oriented application, except that it includes an additional layer for handling network communication when an object resides on a different machine. This additional layer is handled by special objects called stubs and skeletons.




On CORBA clients, the stub acts as a proxy for an object that may be implemented by the same process, another process, or on another (server) machine. The client interacts with the stub as if it were any other object that implements an interface. For more information about using interfaces, see Using interfaces.

However, unlike most objects that implement interfaces, the stub handles interface calls by calling into the ORB software that is installed on the client machine. The VisiBroker ORB uses a Smart Agent (osagent) that is running somewhere on the local area network. The Smart Agent is a dynamic, distributed directory service that locates an available server which provides the real object implementation.

On the CORBA server, the ORB software passes interface calls to an automatically-generated skeleton. The skeleton communicates with the ORB software through the Basic Object Adaptor (BOA). Using the BOA, the skeleton registers the object with the Smart Agent, indicates the scope of the object (whether it can be used on remote machines), and indicates when objects are instantiated and ready to respond to clients.

The following topics describe how you can adapt this basic model to suit the needs of your distributed application:

- Understanding stubs and skeletons
- Using Smart Agents
- Activating server applications
- Binding interface calls dynamically

## Understanding stubs and skeletons

Topic groups See also

Stubs and skeletons provide the mechanism that allows CORBA applications to marshal interface calls. Marshaling

- Takes an interface pointer in the server's process and makes the pointer available to code in the client process.
- Transfers the arguments of an interface call as passed from the client and places the arguments into the remote object's process space.

For any interface call, the caller pushes arguments onto the stack and makes a function call through the interface pointer. If the object is not in the same process space as the code that calls its interface, the call gets passed to a stub which is in the same process space. The stub writes the arguments into a marshaling buffer and transmits the call in a structure to the remote object. The server skeleton unpacks this structure, pushes the arguments onto the stack, and calls the object's implementation. In essence, the skeleton recreates the client's call in its own address space.

Stubs and skeletons are created for you automatically when you define the object's interface. Their definitions are created in the _TLB unit that is created when you define the interface. You can view this unit by selecting it in your implementation unit's **uses** clause and typing *Control-Enter*. For more information about defining the object's interface, see Defining object interfaces.

## Using Smart Agents

Topic groups  See also

The Smart Agent (osagent) is a dynamic, distributed directory service that locates an available server that implements an object. If there are multiple servers to choose from, the Smart Agent provides load balancing. It also protects against server failures by attempting to restart the server when a connection fails, or, if necessary, locating a server on another host.

A Smart Agent must be started on at least one host in your local network, where local network refers to a network within which a broadcast message can be sent. The ORB locates a Smart Agent by using a broadcast message. If the network includes multiple Smart Agents, the ORB uses the first one that responds. Once the Smart Agent is located, the ORB uses a point-to-point UDP protocol to communicate with the Smart Agent. The UDP protocol consumes fewer network resources than a TCP connection.

When a network includes multiple Smart Agents, each Smart Agent recognizes a subset of the objects available, and communicates with other Smart Agents to locate objects it can't recognize directly. If one Smart Agent terminates unexpectedly, the objects it keeps track of are automatically re-registered with another available Smart Agent.

For details about configuring and using Smart Agents on your local networks, see Configuring Smart Agents.

## Activating server applications

Topic groups See also

When the server application starts, it informs the ORB (through the Basic Object Adaptor) of the objects that can accept client calls. This code to initialize the ORB and inform it that the server is up and ready is added to your application automatically by the wizard you use to start your CORBA server application.

Typically, CORBA server applications are started manually. However, you can use the Object Activation Daemon (OAD) to start your servers or instantiate their objects only when clients need to use them.

To use the OAD, you must register your objects with it. When you register your objects with the OAD, it stores the association between your objects and the server application that implements them in a database called the Implementation Repository.

Once there is an entry for your object in the Implementation Repository, the OAD simulates your application to the ORB. When a client requests the object, the ORB contacts the OAD as if it were the server application. The OAD then forwards the client request to the real server, launching the application if necessary.

For details about registering your objects with the OAD, see Registering interfaces with the Object Activation Daemon.

## Binding interface calls dynamically

Topic groups  See also

Typically, CORBA clients use static binding when calling the interfaces of objects on the server. This approach has many advantages, including faster performance and compile-time type checking. However, there are times when you can't know until runtime what interface you want to use. For these cases, Delphi lets you bind to interfaces dynamically at runtime.

Before you can take advantage of dynamic binding, you must register your interfaces with the Interface Repository using the idl2ir utility.

For details on how to use dynamic binding in your CORBA client applications, see Using the dynamic invocation interface.

## Writing CORBA servers

Topic groups See also

Two wizards on the Multi Tier page of the New Items dialog let you create CORBA servers:

- The CORBA Data Module wizard lets you create a CORBA server for a multi-tiered database application.
- The CORBA Object wizard lets you create an arbitrary CORBA server.

In addition, you can easily convert an existing Automation server to a CORBA server by right-clicking and choosing Expose As CORBA Object. When you expose an Automation server as a CORBA object, you create a single application that can service both COM clients and CORBA clients simultaneously.

### Using the CORBA wizards

To start the wizard, choose File|New to display the New Items dialog. Select the Multi Tier page, and double-click the appropriate wizard.

You must supply a class name for your CORBA object. This is the base name of a descendant of *TCorbaDataModule* or *TCorbaImplementation* that your application creates. It is also the base name of the interface for that class. For example, if you specify the class name *MyObject*, the wizard creates a new unit declaring *TMyObject* which implements the *IMyObject* interface.

The wizard lets you specify how you want your server application to create instances of this object. You can choose either shared or instance-per-client.

- When you choose shared, your application creates a single instance of the object that handles all client requests. This is the model used in traditional CORBA development. Because the single object instance is shared by all clients, it must not rely on any persistent state information such as property settings.
- When you choose instance-per-client, a new object instance is created for each client connection. This instance persists until a specified timeout period elapses with no messages from the client. This model lets you use persistent state information, because separate clients can't interfere with each other's property settings. However, client applications must call the server often enough so that the server object does not time out.

**Note:** The instance-per-client model is not typical of most CORBA development, but works, rather, like the COM model, where the lifetime of a server object is governed by client usage. This model allows Delphi to create servers that act as CORBA and COM servers simultaneously.

In addition to the instancing model, you must specify the threading model. You can choose Single- or Multi-threaded.

- If you choose Single-threaded, each object instance is guaranteed to receive only one client request at a time. You can safely access your object's instance data (properties or fields). However, you must guard against thread conflicts when you use global variables or objects.
- If you choose Multi-threaded, each client connection has its own dedicated thread. However, your application may be called by multiple clients simultaneously, each on a separate thread. You must guard against simultaneous access of instance data as well as global memory. Writing Multi-threaded servers is tricky when you are using a shared object instance, because you must protect against thread conflicts for all data and objects contained in your application.

The following topics describe how your server application is created once you have started it using the appropriate Wizard:

- Defining object interfaces
- Automatically generated code
- Registering server interfaces

## Defining object interfaces

Topic groups  See also

With traditional CORBA tools, you must define object interfaces separately from your application, using the CORBA Interface Definition Language (IDL). You then run a utility that generates stub-and-skeleton code from that definition. However, Delphi generates the stub, skeleton, and IDL for you automatically. You can easily edit your interface using the Type Library editor and Delphi automatically updates the appropriate source files.

The Type Library editor is also used for defining COM-based type libraries. Because of this, it includes many options and controls that are not relevant to CORBA applications. If you try to use these options, (for example, if you try to specify a version number or help file), your settings are ignored. If you create a COM Automation server which you then expose as a CORBA server, these settings apply to your server in its role as an Automation server.

In the Type Library editor, you can define your interface using Object Pascal syntax or the Microsoft IDL that is used for COM objects. Specify which language you want to use when defining your interfaces on the Type Library page of the Environment Options dialog. If you choose to use IDL, be aware that the Microsoft IDL differs slightly from the CORBA IDL. When defining your interface, you are limited to the types listed in the following table:

| Type | Details |
|---|---|
| ShortInt | 8-bit signed integer |
| Byte | 8-bit unsigned integer |
| SmallInt | 16-bit signed integer (CORBA short) |
| Word | 16-bit unsigned integer (CORBA unsigned short) |
| Longint, Integer | 32-bit signed integer (CORBA long) |
| Cardinal | 32-bit unsigned integer (CORBA unsigned long) |
| Single | 4-byte floating point value (CORBA float) |
| Double | 8-byte floating point value (CORBA double) |
| TDateTime | Passed as a Double value (CORBA double) |
| PWideChar | Unicode string |
| String, PChar | Strings must be cast to a PChar (passed as CORBA char array) |
| Variant | Passed as a CORBA Any. This is the only way to pass an Array or Currency value. |
| Boolean | passed as a CORBA_Boolean (Byte) |
| Object Reference or interface | Passed as a CORBA interface |
| Enumerated types | Passed as an Integer (CORBA long) |

**Note:** Instead of using the Type Library editor, you can add to your interface by right-clicking in the code editor and choosing Add To Interface. However, you will need to use the Type Library editor to save an .IDL file for your interface.

You can't add properties that use parameters (although you can add get and set methods for such properties). Some types (such as arrays, Int64 values, or Currency types) must be specified as Variants. Records are not supported in the Client/Server version.

Your interface definition is reflected in the automatically generated stub-and-skeleton unit. This unit is updated when you choose Refresh in the type library editor or when you use the Add To Interface command. This automatically generated unit is added to the uses clause of you implementation unit. Do not edit the stub-and-skeleton unit.

In addition to the stub-and-skeleton unit, editing the interface updates your server implementation unit by adding declarations for your interface members and providing empty implementations for the methods. You can then edit this implementation unit to provide meaningful code for the body of each new interface method.

**Note:** You can save a CORBA .IDL file for your interface by clicking the Export button while in the Type Library editor. Specify that the .IDL file should use CORBA IDL, not Microsoft IDL. Use this .IDL file for registering your interface or for generating stubs and skeletons for other languages.

## Automatically generated code

Topic groups  See also

When you define CORBA object interfaces, two unit files are automatically updated to reflect your interface definitions.

The first of these is the stub-and-skeleton unit. It has a name of the form MyInterface_TLB.pas. While this unit defines the stub class that is only used by client applications, it also contains the declaration for your interface types and your skeleton classes. You should not edit this file directly. However, this unit is automatically added to the **uses** clause of your implementation unit.

The stub-and-skeleton unit defines a skeleton object for every interface supported by your CORBA server. The skeleton object is a descendant of *TCorbaSkeleton*, and handles the details of marshaling interface calls. It does not implement the interfaces you define. Instead, its constructor takes an interface instance which it uses to handle all interface calls.

The second updated file is the implementation unit. By default, it has a name of the form unit1.pas, although you will probably want to change this to a more meaningful name. This is the file that you edit.

For each CORBA interface you define, an implementation class definition is automatically added to your implementation unit. The implementation class name is based on the interface name. For example, if the interface is *IMyInterface*, the implementation class is named *TMyInterface*. You will find code added to this class's implementation for every method you add to the interface. You must fill in the body of these methods to finish the implementation class.

In addition, you may notice that some code is added to the initialization section of your implementation unit. This code creates a *TCorbaFactory* object for each object interface you expose to CORBA clients. When clients call your CORBA server, the CORBA factory object creates or locates an instance of your implementation class and passes it as an interface to the constructor for the corresponding skeleton class.

**Note:** The use of factory objects that indirectly create your CORBA server objects is not typical of most CORBA development. Instead, it follows the model used by COM server applications, and enables Delphi to build servers that act as COM and CORBA servers simultaneously. Factories allow CORBA servers to implement the instance-per-client model. Clients of CORBA servers built with Delphi use the *CorbaFactoryCreateStub* function, which handles the details of instructing the factory on the CORBA server to create a CORBA object.

## Registering server interfaces

Topic groups  See also

While it is not necessary to register your server interfaces if you are only using static binding of client calls into your server objects, registering your interfaces is recommended. There are two utilities with which you can register your interfaces:

- **The Interface Repository.** By registering with the Interface Repository, clients can take advantage of dynamic binding. This allows your server to respond to clients that are not written in Delphi if they use the dynamic invocation interface (DII). Registering with the Interface Repository is also a convenient way to allow other developers to view your interfaces when they write client applications.
- **The Object Activation Daemon.** By registering with the Object Activation Daemon (OAD), your server need not be launched or your objects instantiated until they are needed by clients. This conserves resources on your server system.

## Registering interfaces with the Interface Repository

Topic groups  See also

You can create an Interface Repository for your interfaces by running the Interface Repository server. First, you must save the .IDL file for your interface. To do this, choose View|Type Library, and then, in the Type Library Editor, click the Export button to export your interface as a CORBA .IDL file.

Once you have an .IDL file for your interface, you can run the Interface Repository Server using the following syntax:

```
irep [-console] IRname [file.idl]
```

The irep arguments are described in the following table:

| Argument | Description |
|---|---|
| -console | Starts the Interface Repository server as a console application. By default, the Interface Repository server runs as a Windows application. |
| IRname | The name of the Interface Repository. While the server is running, clients use this name to bind to the Interface Repository so that they can obtain interface information for DII or so that they can register additional interfaces. |
| file.idl | An .IDL file that describes the initial contents of the Interface Repository. If you do not specify a file name, the Interface Repository starts out empty. You can subsequently add interfaces using the menus of the Interface Repository server, or using the idl2ir utility. |

Once the Interface Repository server is running, you can add additional interfaces by choosing File|Load and specifying a new .IDL file. However, if the new .IDL file contains any entries that match an existing .IDL entry, the new .IDL file is rejected.

At any point, you can save the current contents of the Interface Repository to an .IDL file by choosing File|Save or File|Save As. This way, after you exit the Interface Repository, you can restart it later with the saved file so that you don't need to reimport all changes to the initial .IDL file.

You can also register additional interfaces with the Interface Repository using the idl2ir utility. While the Interface Repository server is running, start the idl2ir utility using the following syntax:

```
idl2ir [-ir IRname] {-replace} file.idl
```

The idl2ir arguments are described in the following table:

| Argument | Description |
|---|---|
| -ir IRname | Directs the utility to bind to the interface repository instance named IRname. If this argument is not specified, idl2ir binds to any interface repository returned by the smart agent. |
| -replace | Directs the utility to replace interface repository items with matching items in file.idl. If -replace is not specified, the entire interface is added to the repository, unless there are matching items, in which case the utility rejects the entire .IDL file. If -replace is specified, non-matching items are rejected. |
| file.idl | Specifies the .IDL file that contains the updates for the Interface Repository. |

Entries in an interface repository can't be removed while the Interface Repository server is running. To remove an item, you must shut down the Interface Repository server, generate a new .IDL file, and then start the Interface Repository server, specifying the updated .IDL file.

## Registering interfaces with the Object Activation Daemon

Topic groups  See also

Before you can register an interface with the Object Activation Daemon (OAD), the OAD command-line program must be running on at least one machine on your local network. Start the OAD using the following syntax:

```
oad [options]
```

The OAD utility accepts the following command line arguments:

| Argument | Description |
|---|---|
| -v | Turns on verbose mode. |
| -f | Stipulates that the process should not fail if another OAD is running on this host. |
| -t<n> | Specifies the number of seconds the OAD will wait for a spawned server to activate the requested object. The default time-out is 20 seconds. Setting this value to 0 causes the OAD to wait indefinitely. If the spawned server process does not activate the requested object within the time-out period, the OAD terminates the server process and returns an exception. |
| -C | Allows the OAD to run in console mode if it has been installed as an NT service. |
| -k | Stipulates that an object's child process should be killed once all of its objects are unregistered with the OAD. |
| -? | Describes these arguments. |

Once the OAD is running, you can register your object interfaces using the command-line program oadutil. First, you must export the .IDL file for your interfaces. To do this, click the Export button in the Type Library editor and save the interface definition as a CORBA .IDL file.

Next, register interfaces using the oadutil program with the following syntax:

```
oadutil reg [options]
```

The following arguments are available when registering interfaces using oadutil:

| Arguments | Description |
|---|---|
| -i <interface name> | Specifies a particular IDL interface name. You must specify the interface to register using either this or the -r option. |
| -r <repository id> | Specifies a particular interface by its repository id. The repository id is a unique identifier associated with the interface. You must specify the interface to register using this or the -i option. |
| -o <object name> | Specifies the name of the object that supports the interface. This option is required. |
| -cpp <file name> | Specifies the name of your server executable. This option is required. |
| -host <host name> | Specifies a remote host where the OAD is running. (optional) |
| -verbose | Starts the utility in verbose mode. Messages are sent to stdout. (optional) |
| -d<reference data> | Specifies reference data that is passed to the server application on activation. (optional) |
| -a arg1 | Specifies command-line arguments that are passed to the server application. Multiple arguments can be passed with multiple -a parameters. (optional) |
| -e env1 | Specifies environment variables that are passed to the server application. Multiple arguments are passed with multiple -e parameters. (optional) |

For example, the following line registers an interface based on its repository id:

```
oadutil reg -r IDL:MyServer/MyObjectFactory:1.0 -o TMyObjectFactory -cpp
MyServer.exe -p unshared
```

**Note:** You can obtain the repository ID for your interface by looking at the code added to the initialization section of your implementation unit. It appears as the third argument of the call to *TCorbaFactory.Create*.

When an interface becomes unavailable, you must unregister it. Once an object is unregistered, it can no longer be automatically activated by the OAD if a client requests the object. Only objects that have been previously registered using oadutil reg can be unregistered.

To unregister interfaces, use the following syntax:

```
oadutil unreg [options]
```

The following arguments are available when unregistering interfaces using oadutil:

| Argument | Description |
|---|---|
| -i <interface name> | Specifies a particular IDL interface name. You must specify the interface to unregister using either this or the -r option. |
| -r <repository id> | Specifies a particular interface by its repository id. The repository id is a unique identifier associated with the interface when it is registered with the Interface Repository. You must specify the interface to unregister using this or the -i option. |
| -o <object name> | Specifies the name of the object that supports the interface. If you do not specify the object name, all objects that support the specified interface are unregistered. |
| -host <host name> | Specifies a remote host where the OAD is running. (optional) |
| -verbose | Starts the utility in verbose mode. Messages are sent to stdout. (optional) |
| -version | Displays the version number for oadutil. |

## Writing CORBA clients

Topic groups  See also

When you write a CORBA client, the first step is to ensure that the client application can talk to the ORB software on the client machine. To do this, simply add CorbaInit to the **uses** clause of your unit file.

Next, proceed with writing your application in the same way you write any other application in Delphi. However, when you want to use objects that are defined in the server application, you do not work directly with an object instance. Instead, you obtain an interface for the object and work with that. You can obtain the interface in one of two ways, depending on whether you want to use static or dynamic binding.

To use static binding, you must add a stub-and-skeleton unit to your client application. The stub-and-skeleton unit is created automatically when you save the server interface. Using static binding is faster than using dynamic binding, and provides additional benefits such as compile-time type checking and code-completion.

However, there are times when you do not know until runtime what object or interface you want to use. For these cases, you can use dynamic binding. Dynamic binding does not require a stub unit, but it does require that all remote object interfaces you use are registered with an Interface Repository running on the local network.

**Tip:** You may want to use dynamic binding when writing CORBA clients for servers that are not written in Delphi. This way, you do not need to write your own stub class for marshaling interface calls.

## Using stubs

Topic groups  See also

Stub classes are generated automatically when you define a CORBA interface. They are defined in a stub-and-skeleton unit, which has a name of the form BaseName_TLB (in a file with a name of the form BaseName_TLB.pas).

When writing a CORBA client, you do not edit the code in the stub-and-skeleton unit. Add the stub-and-skeleton unit to the uses clause of the unit which needs an interface for an object on the CORBA server.

For each server object, the stub-and-skeleton unit contains an interface definition and a class definition for a corresponding stub class. For example, if the server defines an object class *TServerObj*, the stub-and-skeleton unit includes a definition for the interface *IServerObj*, and for a stub class *TServerObjStub*. The stub class is a descendant of *TCorbaDispatchStub*, and implements its corresponding interface by marshaling calls to the CORBA server. In addition to the stub class, the stub-and-skeleton unit defines a stub factory class for each interface. This stub factory class is never instantiated: it defines a single class method.

In your client application, you do not directly create instances of the stub class when you need an interface for the object on the CORBA server. Instead, call the class method *CreateInstance* of the stub factory class. This method takes one argument, an optional instance name, and returns an interface to the object instance on the server. For example:

```
var
  ObjInterface : IServerObj;
begin
  ObjInterface := TServerObjFactory.CreateInstance('');
  ...
end;
```

When you call *CreateInstance*, it

1. Obtains an interface instance from the ORB.
2. Uses that interface to create an instance of the stub class.
3. Returns the resulting interface.

**Note:** If you are writing a client for a CORBA server that was not written using Delphi, you must write your own descendant of *TCorbaStub* to provide marshaling support for your client. You must then register this stub class with the global *CORBAStubManager*. Finally, to instantiate the stub class and get the server interface, you can call the global *BindStub* procedure to obtain an interface which you then pass to the CORBA stub manager's *CreateStub* method.

## Using the dynamic invocation interface

Topic groups  See also

The dynamic invocation interface (DII) allows client applications to call server objects without using a stub class that explicitly marshals interface calls. Because DII must encode all type information before the client sends a request and then decode that information on the server, it is slower than using a stub class.

Before you can use DII, the server interfaces must be registered with an Interface Repository that is running on the local network.

To use DII in a client application, obtain a server interface and assign it to a variable of type *TAny*. *TAny* is a special, CORBA-specific Variant. Then call the methods of the interface using the *TAny* variable as if it were an interface instance. The compiler handles the details of turning your calls into DII requests.

## Obtaining the interface

Topic groups  See also

To get an interface for making late-bound DII calls, use the global *CorbaBind* function. *CorbaBind* takes either the Repository ID of the server object or an interface type. It uses this information to request an interface from the ORB, and uses that to create a stub object.

**Note:** Before calling *CorbaBind*, the association between the interface type and its Repository ID must be registered with the global *CorbaInterfaceIDManager*.

If your client application has a registered stub class for the interface type, *CorbaBind* creates a stub of that class. In this case, the interface returned by *CorbaBind* can be used for both early binding (by casting with the **as** operator) or late (DII) binding. If there is no registered stub class for the interface type, *CorbaBind* returns the interface to a generic stub object. A generic stub object can only be used for late (DII) calls.

To use the interface returned by *CorbaBind* for DII calls, assign it to a variable of type *TAny*:

```
var
  IntToCall: TAny;
begin
  IntToCall := CorbaBind('IDL:MyServer/MyServerObject:1.0');
  ...
```

## Calling interfaces with DII

Topic groups  See also

Once an interface has been assigned to a variable of type *TAny*, calling it using DII simply involves using the variable as if it were an interface:

```
var
  HR, Emp, Payroll, Salary: TAny;
begin
  HR := CorbaBind('IDL:CompanyInfo/HR:1.0');
  Emp := HR.LookupEmployee(Edit1.Text);
  Payroll := CorbaBind('IDL:CompanyInfo/Payroll:1.0');
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit2.Text) / 100));
end;
```

When using DII, all interface methods are case sensitive. Unlike when making statically-bound calls, you must be sure that method names match the case used in the interface definition.

When calling an interface using DII, every parameter is treated as a value of type *TAny*. This is because *TAny* values carry their type information with them. This type information allows the server to interpret type information when it receives the call.

Because the parameters are always treated as *TAny* values, you do not need to explicitly convert to the appropriate parameter type. For example, in the previous example, you could pass a string instead of a floating-point value for the last parameter in the call to *SetEmployeeSalary*:

```
Payroll.SetEmployeeSalary(Emp, Edit2.Text);
```

You can always pass simple types directly as parameters, and the compiler converts them to *TAny* values. For structured types, you must use the conversion methods of the global ORB variable to create an appropriate *TAny* type. The following table indicates the method to use for creating different structured types:

| Structured type | helper function |
|---|---|
| record | MakeStructure |
| array (fixed length) | MakeArray |
| dynamic array (sequence) | MakeSequence |

When using these helper functions, you must specify the type code that describes the type of record, array, or sequence you want to create. You can obtain this type dynamically from a Repository ID using the ORB's *FindTypeCode* method:

```
var
  HR, Name, Emp, Payroll, Salary: TAny;
begin
  with ORB do
  begin
    HR := Bind('IDL:CompanyInfo/HR:1.0');
    Name := MakeStructure(FindTypeCode('IDL:CompanyInfo/EmployeeName:1.0',
                                       [Edit1.Text,Edit2,Text]);
    Emp := HR.LookupEmployee(Name);
    Payroll := Bind('IDL:CompanyInfo/Payroll:1.0');
  end;
  Salary := Payroll.GetEmployeeSalary(Emp);
  Payroll.SetEmployeeSalary(Emp, Salary + (Salary * StrToInt(Edit3.Text) / 100));
end;
```

## Customizing CORBA applications

Topic groups  See also

Two global functions, ORB and BOA, allow you to customize the way your application interacts with the CORBA software running on your network.

Client applications use the value returned by ORB to configure the ORB software, disconnect from the server, bind to interfaces, and obtain string representations for objects so that they can display object names in the user interface.

Server applications use the value returned by BOA to configure the BOA software, expose or hide objects, and retrieve custom information assigned to an object by client applications.

The following topics provide additional details on how to perform some of these tasks:

- Displaying objects in the user interface
- Exposing and hiding CORBA objects
- Passing client information to server objects

## Displaying objects in the user interface

Topic groups  See also

When writing a CORBA client application, you may wish to present users with the names of available CORBA server objects. To do this, you must convert your object interface to a string. The *ObjectToString* method of the global ORB variable performs this conversion. For example, the following code displays the names of three objects in a list box, given interface instances for their corresponding stub objects.

```
var
  Dept1, Dept2, Dept3: IDepartment;
  begin
  Dept1 := TDepartmentFactory.CreateInstance('Sales');
 Dept1.SetDepartmentCode(120);
  Dept2 := TDepartmentFactory.CreateInstance('Marketing');
  Dept2.SetDepartmentCode(98);
  Dept3 := TSecondFactory.CreateInstance('Payroll');
  Dept3.SetDepartmentCode(49);
ListBox1.Items.Add(ORB.ObjectToString(Dept1));
  ListBox1.Items.Add(ORB.ObjectToString(Dept2));
  ListBox1.Items.Add(ORB.ObjectToString(Dept3));
end;
```

The advantage of letting the ORB create strings for your objects is that you can use the *StringToObject* method to reverse this procedure:

```
var
  Dept: IDepartment;
begin
  Dept := ORB.StringToObject(ListBox1.Items[ListBox1.ItemIndex]);
  ... { do something with the selected department }
```

## Exposing and hiding CORBA objects

Topic groups  See also

When a CORBA server application creates an object instance, it can make that object available to clients by calling the *ObjIsReady* method of the global variable returned by BOA.

Any object exposed using *ObjIsReady* can be hidden by the server application. To hide an object, call the BOA's *Deactivate* method.

If the server deactivates an object, this can invalidate an object interface held by a client application. Client applications can detect this situation by calling the stub object's *NonExistent* method. *NonExistent* returns *True* when the server object has been deactivated and *False* if the server object is still available.

## Passing client information to server objects

Topic groups  See also

Stub objects in CORBA clients can send identifying information to the associated server object using a *TCorbaPrincipal*. A *TCorbaPrincipal* is an array of bytes that represents information about the client application. Stub objects set this value using their *SetPrincipal* method.

Once the CORBA client has written principal data to the server object instance, the server object can access this information using the BOA's *GetPrincipal* method.

Because *TCorbaPrincipal* is an array of bytes, it can represent any information that the developer finds useful to send. For example, clients with special privileges can send a key value that the server checks before making some methods available.

## Deploying CORBA applications

Topic groups  See also

Once you have created client or server applications and thoroughly tested them, you are ready to deploy client applications to end users' desktops and server applications on server-class machines. The following list describes the files that must be installed (in addition to your client or server application) when you deploy your CORBA application:

- The ORB libraries must be installed on every client and server machine. These libraries are found in the Bin subdirectory of the directory where you installed VisiBroker.
- If your client uses the dynamic invocation interface (DII), you must run an Interface Repository server on at least one host on the local network, where local network refers to a network within which a broadcast message can be sent. See Registering interfaces with the Interface Repository for details on how to run the Interface Repository server and register the appropriate interfaces.
- If your server should be started on demand, the Object Activation Daemon (OAD) must be running on at least one host on the local network. See Registering interfaces with the Object Activation Daemon for details on how to run the OAD and register the appropriate interfaces.
- A Smart Agent (osagent) must be installed on at least one host on the local network. You may want to deploy the Smart Agent to more than one machine. See Configuring Smart Agents for more information.

In addition, you may need to set the following environment variables when you deploy your CORBA application.

| Variable | Meaning |
|---|---|
| PATH | You must ensure that the directory which contains the ORB libraries is on the Path. |
| VBROKER_ADM | Specifies the directory that contains configuration files for the Interface Repository, Object Activation Daemon, and Smart Agent. |
| OSAGENT_ADDR | Specifies the IP address of the host machine whose Smart Agent should be used. If this variable is not set, the CORBA application uses the first Smart Agent that answers a broadcast message. |
| OSAGENT_PORT | Specifies the port which a Smart Agent uses to listen for requests. |
| OSAGENT_ADDR_FILE | Specifies the fully qualified path name to a file containing Smart Agent addresses on other local networks. |
| OSAGENT_LOCAL_FILE | Specifies the fully qualified path name to a file containing network information for a Smart Agent running on a multi-homed host. |
| VBROKER_IMPL_PATH | Specifies the directory for the Implementation Repository (where the OAD stores its information). |
| VBROKER_IMPL_NAME | Specifies the default file name for the Implementation Repository. |

**Note:** For general information about deploying applications, see Deploying applications.

## Configuring Smart Agents

Topic groups

When you deploy your CORBA application, there must be at least one smart agent running on the local network. By deploying more than one smart agent on a local network, you provide protection against the machine that is running the smart agent from going down.

You deploy smart agents so that they divide a local network into separate ORB domains. Conversely, you can connect smart agents on different local networks to broaden the domain of your ORB.

The following topics describe how to configure and use smart agents on your local network:

- Starting the Smart Agent
- Configuring ORB domains
- Connecting Smart Agents on different local networks

## Starting the Smart Agent

Topic groups  See also

To start the Smart Agent, run the osagent utility. You must run at least one Smart Agent on a host in your local network. The osagent utility accepts the following command line arguments:

| Argument | Description |
|---|---|
| -v | Turns on verbose mode. Information and diagnostic messages are written to a log file named osagent.log. You can find this file in the directory specified by the VBROKER_ADM environment variable. |
| -p<n> | Specifies the UDP port that the Smart Agent uses to listen for broadcast messages. |
| -C | Allows the Smart Agent to run in console mode if it has been installed as an NT service. |

For example, type the following command in a DOS box or choose Run from the Start button:

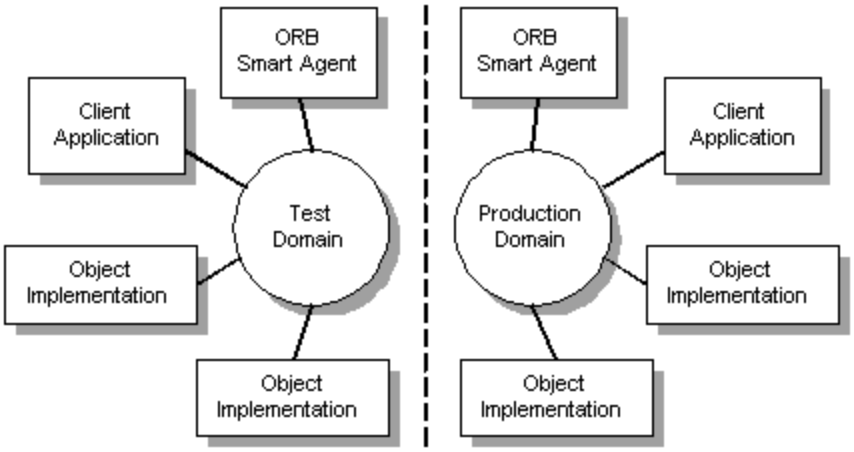
```
osagent -p 11000
```

This starts the Smart Agent so that it listens on UDP port 11000 rather than the default port (14000). Changing the port which the Smart Agent uses to listen for broadcast messages allows you to create multiple ORB domains.

## Configuring ORB domains

Topic groups  See also

It is often desirable to have two or more separate ORB domains running at the same time. One domain might consist of the production version of client applications and object implementations while another domain could include test versions of the same clients and objects that have not yet been released for general use. If several developers are working on the same local network, each may want a dedicated ORB domain so that the testing efforts of different developers do not conflict with each other.




You can distinguish between two or more ORB domains on the same network by using a unique UDP port number of the osagents in each domain.

The default port number (14000) is written to the Windows Registry when the ORB is installed. To override this value, set the OSAGENT_PORT environment variable to a different setting. You can further override the value specified by OSAGENT_PORT by starting the Smart Agent using the -p option.

## Connecting Smart Agents on different local networks

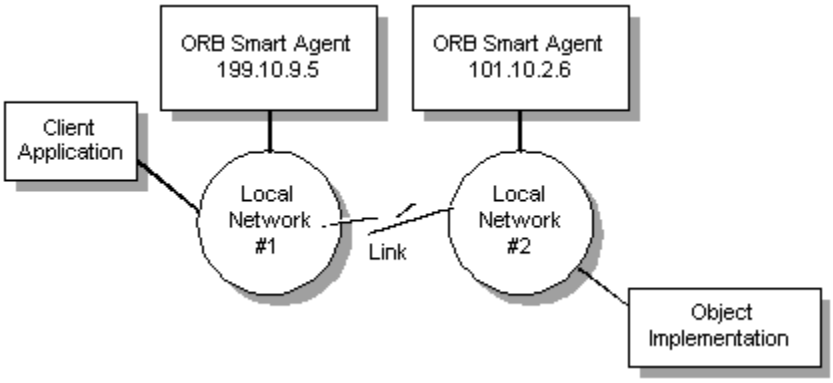
Topic groups  See also

If you start multiple Smart agents on your local network, they will discover each other using UDP broadcast messages. Your local networks are configured by specifying the scope of broadcast messages using the IP subnet mask. You can allow a Smart Agent to communicate with other networks in two ways:

- Using an agentaddr file.
- Using a multi-homed host.

### Using an agentaddr file

Consider the two Smart Agents depicted in the following figure. The Smart Agent on network #1 listens for broadcast messages using the IP address 199.10.9.5. The Smart Agent on network #2 listens using the IP address 101.10.2.6.




The Smart Agent on network #1 can contact the Smart Agent on network #2 if it can find a file named **agentaddr** which contains the following line:

```
101.10.2.6
```

The Smart Agent looks for this file in the directory specified by the VBROKER_ADM environment variable.

### Using a multi-homed host

When you start the Smart Agent on a host that has more than one IP address (known as a multi-homed host), it can provide a powerful mechanism for bridging objects located on separate local networks. All local networks to which the host is connected can communicate with a single Smart Agent, effectively bridging the local networks without an **agentaddr** file.

However, on a multi-homed host, the Smart Agent can't determine the correct subnet mask and broadcast address values. You must specify these values in a **localaddr** file. You can obtain the appropriate network interface values of the **localaddr** file by accessing the TCP/IP protocol properties from the Network Control Panel. If your host is running Windows NT, you can use the ipconfig command to get these values.

The **localaddr** file contains a line for every combination of IP address, subnet mask, and broadcast address that the Smart Agent can use. For example, the following lists the contents of a **localaddr** file for a Smart Agent that has two IP addresses:

```
216.64.15.10 255.255.255.0 216.64.15.255
214.79.98.88 255.255.255.0 214.79.98.255
```

You must also set the OSAGENT_LOCAL_FILE environment variable to the fully qualified path name of the **localaddr** file. This allows the Smart Agent to locate this file.

## Related topic groups

### Developing Internet applications

- Creating Internet server applications
- Action items
- Creating web server applications
- Using page producer components
- Debugging web server applications
- The Web dispatcher
- Client requests
- URL parts
- Client requests
- HTTP responses
- Web server application structure
- Working with sockets
- Socket components
- Describing hosts and using ports
- Socket events
- Socket I/O
- Service protocols and ports
- Connection types
- Using CORBA with Delphi

## Creating Internet server applications

Related topic groups

- Creating Internet server applications: Overview
- Terminology and standards
- HTTP server activity
- Web server applications
- Using predefined HTML-transparent tag names
- Debugging server applications

## Action items

Related topic groups

- Determining when action items fire
- The target URL
- The request method type
- Enabling and disabling action items
- Choosing a default action item
- Responding to request messages with action items

## Creating web server applications

Related topic groups

- Types of Web server applications
- Creating Web server applications
- The Web module
- The Web Application object

## Using page producer components

Related topic groups

- Using page producer components
- HTML templates
- Specifying the HTML template
- Converting HTML-transparent tags
- Using page producers from an action item
- Chaining page producers together
- Using database information in responses
- Adding a session to the Web module
- Representing a dataset in HTML
- Using dataset page producers
- Using table producers
- Specifying the table attributes
- Specifying the row attributes
- Specifying the columns
- Embedding tables in HTML documents
- Using TDataSetTableProducer
- Using TQueryTableProducer

## Debugging web server applications

Related topic groups

- Debugging ISAPI and NSAPI applications
- Debugging with a Microsoft IIS server
- Debugging with a Windows 95 Personal Web Server
- Debugging with Netscape Server Version 2.0
- Debugging CGI and Win-CGI applications

## The Web dispatcher

Related topic groups

- Adding actions to the dispatcher
- Dispatching request messages

## Client requests

Related topic groups

- Composing client requests
- Serving client requests
- Responding to client requests

## URL parts

Related topic groups

- Parts of a Uniform Resource Locator
- HTTP request header information

## Client requests

Related topic groups

- Properties that contain request header information
- Properties that identify the target
- Properties that describe the Web client
- Properties that identify the purpose of the request
- Properties that describe the expected response
- Properties that describe the content
- The content of HTTP request messages

## HTTP responses

Related topic groups

- Filling in the response header
- Indicating the response status
- Indicating the need for client action
- Describing the server application
- Describing the content
- Setting the response content
- Sending the response

## Web server application structure

Related topic groups

- The structure of a Web server application
- The Web dispatcher
- Action items
- Accessing client request information
- Creating HTTP response messages
- Generating the content of response messages

## Working with sockets

Related topic groups

- Working with sockets
- Implementing services
- Types of socket connections
- Describing sockets
- Using socket components
- Responding to socket events

## Socket components

Related topic groups

- Using client sockets
- Specifying the desired server
- Forming the connection
- Getting information about the connection
- Closing the connection
- Using server sockets
- Specifying the desired server
- Listening for client requests
- Connecting to clients
- Getting information about connections
- Closing server connections

## Describing hosts and using ports

Related topic groups

- Describing the host
- Using ports

## Socket events

Related topic groups

- Error events
- Client events
- Server events

## Socket I/O

Related topic groups

- Reading and writing over socket connections
- Non-blocking connections
- Reading and writing events
- Blocking connections
- Using threads with blocking connections
- Using TWinSocketStream
- Writing client threads
- Writing server threads

## Service protocols and ports

Related topic groups

- Service protocols
- Services and ports

## Connection types

Related topic groups

- Client connections
- Listening connections
- Server connections

## Using CORBA with Delphi

Related topic groups

- Writing CORBA applications
- Overview of a CORBA application
- Understanding stubs and skeletons
- Using Smart Agents
- Activating server applications
- Binding interface calls dynamically
- Writing CORBA servers
- Defining object interfaces
- Automatically generated code
- Registering server interfaces
- Registering interfaces with the Interface Repository
- Registering interfaces with the Object Activation Daemon
- Writing CORBA clients
- Using stubs
- Using the dynamic invocation interface
- Obtaining the interface
- Calling interfaces with DII
- Customizing CORBA applications
- Displaying objects in the user interface
- Exposing and hiding CORBA objects
- Passing client information to server objects
- Deploying CORBA applications
- Configuring Smart Agents
- Starting the Smart Agent
- Configuring ORB domains
- Connecting Smart Agents on different local networks

**Link not found**

The topic you requested is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.

Borland INPRISE

The topic you requested is now loading. If it does not appear within a few seconds, the topic is either not available or not linked to this Help system. This can occur if you launched this Help file from a system on which Delphi has not yet been installed, or if the subject matter you are requesting is not available in your edition of Delphi.