



## Using Siena Services

Version 1.0.2

Copyright © 2002 by Siena Technology Ltd, UK.

All rights reserved. No part of this work may be reproduced or used in any form or by any means without written permission of Siena Technology Ltd.

All product names and logos are trademarks of their respective owners. The product names "Siena Server", "Siena Client" and "Siena Technology" and corresponding logos are trademarks of Siena Technology Ltd.

---

# Contents

<b>1.</b>	<b>Introduction.....</b>	<b>7</b>
1.1	Services - A Primer.....	7
	XML-RPC.....	8
	SOAP .....	8
<b>2.</b>	<b>Introduction to Siena .....</b>	<b>9</b>
2.1	The Basics.....	9
2.2	Structure.....	9
	2.2.1 Server side overview .....	9
	2.2.2 Client side overview.....	10
2.3	Client / Server Interaction: An Example .....	11
<b>3.</b>	<b>Client side set up.....</b>	<b>15</b>
3.1	Configuration.....	15
3.2	Licensing.....	15
3.3	Using the Siena COM interface.....	15
	3.3.1 The useformat method.....	16
3.4	Language specific details.....	17
	3.4.1 Visual Basic Clients .....	18
	3.4.2 Delphi Clients .....	18
3.5	Developer Dashboard.....	19
<b>4.</b>	<b>Server Side Setup.....</b>	<b>20</b>
4.1	Team and Personal Server Configuration.....	20
4.2	Enterprise Server Configuration.....	20
4.3	Installation of User Services .....	25
	4.3.1 The Lib/UserServices/ Directory .....	25

4.3.2	Managing Service Modules Access Permissions.....	26
4.3.3	Updating the Service Registry .....	26
<b>5.</b>	<b>Writing Services .....</b>	<b>28</b>
5.1	Concepts.....	28
5.1.1	Writing Services should be easy.....	28
5.1.2	Publishing Services should be as simple as possible .....	29
5.2	Naming Conventions.....	30
5.2.1	Naming Scheme .....	30
5.2.2	Special Service Namespace.....	30
5.2.3	Service and Method Name Lookup .....	31
5.2.4	Service Name Scopes.....	31
Internal Service Names .....	31	
External Service Names.....	32	
Mapping External to Internal Names .....	32	
5.2.5	Service Parameter Names .....	33
5.3	Organizing Service Implementations .....	33
5.3.1	Services Directory and Auto-Registration of Services .....	33
5.3.2	Registration Error Handling.....	34
5.3.3	Importing Python Modules .....	35
5.4	Service Implementations .....	35
5.4.1	Standard Services .....	36
<b>6.</b>	<b>Domain Services .....</b>	<b>37</b>
6.1.1	Reserved Service and Method Names.....	37
6.1.2	Examples .....	37
Domain methods: domain.method.....	38	
Service methods: domain.service.method .....	38	
Sub-domain method: domain.subdomain.method.....	38	
Sub-domain service method: domain.subdomain.service.method .....	38	
<b>7.</b>	<b>Siena options .....</b>	<b>39</b>
7.1	Server synchronisation .....	39

<b>8.</b>	<b>Service API Reference .....</b>	<b>41</b>
8.1	“Siena.Server.API.Service” Module.....	41
	Class "Error" .....	41
	Class "CancelError" .....	41
	Class "AccessForbiddenError" .....	42
	Class "Service" .....	42
	Attributes: .....	42
	Methods: .....	42
	Reserved Symbols for this Class .....	44
	Class "DomainService" .....	45
	Methods: .....	45
	Reserved Symbols for this Class .....	45
8.2	Data Types.....	46
	8.2.1 COM Mapping.....	46
	8.2.2 Java Mapping.....	46
	8.2.3 Unhandled Types .....	47
8.3	Exceptions and Fault Codes .....	47
	8.3.1 XML-RPC Fault Codes.....	47
8.4	Standard “server.*” Domain Services and Methods .....	49
8.5	Examples.....	50
	8.5.1 Simple implementations.....	50
	Defining a Standard Service Class .....	50
	Get the current date and time .....	51
	Upper case a string .....	51
	Scientific calculations are possible.....	51
	Simply math.....	51
	Standard I/O is possible too.....	52
	8.5.2 More complex implementations.....	52
	Reach out to the Internet.....	52
	Read data from an attached database .....	52
	Implement a Domain Service .....	53
8.6	Debugging User Services .....	53

8.6.1	Server Side Debugging.....	54
	System Log File.....	54
	Printing to the Log File.....	54
	Service Debug Mode .....	55
8.6.2	Client Side Debugging .....	55
	Break Points.....	55
	Logging to the session object.....	55
	Logging to an attached database .....	56
	Client-side development environment .....	56
<b>9.</b>	<b>Frequently Asked Questions (FAQ) .....</b>	<b>57</b>
9.1	Services.....	57
9.1.1	None of the installed services work anymore and the system.log contains a note about UserServices. How can this be fixed ? .....	57
9.1.2	My service xyz doesn't work. What can I do ?.....	57
9.1.3	How do I create a new service domain ? .....	57
<b>10.</b>	<b>Obtaining more information .....</b>	<b>59</b>
<b>Appendix A.....</b>	<b>.....</b>	<b>60</b>
	Connection objects .....	60
	Methods: .....	60
	Cursor objects .....	60
	Attributes:.....	60
	Methods: .....	61

---

## 1. Introduction

Siena provides everything needed to build a powerful service infrastructure easily.

The next section gives you a short introduction to services and the protocols and formats involved. If you are already familiar with those terms, you can safely skip the section and read on.

---

### 1.1 Services - A Primer

In simple words, services are to computers what web pages are to humans: information resources and interaction platforms.

Unlike web pages, which format the data in a human accessible way and provide interaction using forms or graphically using GUI-based techniques, services aim at easy interoperability between computers. The main focus is on communicating programming objects and invoking methods remotely, thereby allowing computers to interact with only very little knowledge about their interfaces.

The service model began with Remote Procedure Calls (RPCs) sometime in the 1970s, but weren't really popular due to the complexity involved in getting them to work, problems with portability and the reduced set of possible parameter types.

Later on, more profound techniques were developed, such as CORBA, which allow sending complex object types across the wire. Interoperability remains a problem, though, and usage is even more complex than that of the early RPC implementations.

With the arrival of the text-based meta language XML, new methods of serializing objects and communicating this information across networks became possible. The text nature of XML makes interoperability much easier to manage than the various binary formats used in previous designs.

Today, there are two popular techniques for invoking methods on remote servers based on XML: XML-RPC and SOAP.

## XML-RPC

XML-RPC is basically a revival of RPC technology and very well suited for working with simple data types and structures. It supports all common types, e.g. integers, floats, datetime, binary and strings. The standard is well defined and there are many implementations of the protocol for all common programming languages. See <http://www.xmlrpc.org/> for more background information.

Siena comes with a standard implementation of XML-RPC.

## SOAP

SOAP was designed to serialize complex object structures and ship the information across a network in a XML format. The object types themselves are user-definable and can be extended as needed. Much like CORBA, SOAP allows to send data in strongly types forms. However, unlike CORBA, SOAP also implements late binding where all the type information is included in the object serialization.

**Note:**

SOAP and WSDL support is available as optional add-on.

---

## 2. Introduction to Siena

---

### 2.1 The Basics

Siena is designed to let you deliver a comprehensive distributed service system within your organisation without requiring your staff to have specialist programming or systems expertise beyond that normally expected of a competent applications programmer.

Installing distributed services helps to enable organisations to manage the growing complexity of maintaining and supporting multiple, duplicated applications on desktops and server computers, while putting the processing of elements of those applications on the most appropriate machine(s).

However, in many frameworks, this is seen to be a difficult and arduous task, both for development and deployment. Siena is designed to make both of these easy to understand and use, allowing application developers to develop and deploy services quickly and efficiently.

---

### 2.2 Structure

The basic architecture of Siena is in two parts: server and client. The server side manages and runs services. The client side provides simple access to running services, with client interfaces available in all major application development languages.

---

#### 2.2.1 Server side overview

There are three Siena servers: Enterprise, Team and Personal.

Enterprise servers run on all machines in the Unix family. They are the full implementation of the Siena server, and would be used in all large applications, with the Siena client interface deployed on client desktops. Three main features differentiate the Enterprise from the Team and Personal

servers. Failover support, essential in any business critical application, is fully implemented in the Enterprise. Secondly, this server acts as a service distribution hub which synchronises with client machines to keep client services up to date automatically. Finally the Enterprise has the capacity to retain session state information for clients.

Team and Personal servers run on any of the Windows or Unix platforms. The Personal server works only on the same machine as the client, and is only accessible from that machine. Team has the same functionality as the Personal, and is accessible to clients on other machines.

All three servers have *Developer* editions which have the full functionality of each respective server and have debugging through a developer dashboard. The dashboard provides diagnostic trace information to the developer's desktop from the Siena server and client modules.

It is not unusual to have a number of Siena servers installed within an organisation – developers are likely to have Personal, Team and Enterprise development servers. The application may demand failover functionality, requiring use of the Enterprise server. Certain services may benefit the system by being run locally, using the client desktop's processing power (for example algorithmic services which are processor intensive and do not require access to data on the main server). In such cases, the client would have a Personal server. Siena provides and manages these aspects easily and intuitively through the design of its distributed architecture.

---

### 2.2.2 Client side overview

For Microsoft clients, Siena provides clientside access through COM. Therefore any programming language that has a COM interface can use Siena in a simple way. Visual Basic, Visual C++ and Delphi are such languages.

Unix clients will use Siena through importing a SienaClient Java class library.

On any platform and development environment, Siena provides a simple application interface for calling services, as shown in the examples below.

Demo version: time expiring version of the personal development server.

---

## 2.3 Client / Server Interaction: An Example

Services are made from a collections of related function calls, called service methods. Each method has a name and a list of parameters, and are called by client programs. These services reside on one or more Siena servers, and are called from client programs - typically on Microsoft Windows from Visual Basic, Delphi or suchlike, and on the Unix family of operating systems from Java, C++, Python etc. And the interface to Siena from such languages could not be easier.

Suppose we have a service called "Invoice", whose rather simple (and not really sensible) collection of methods is ("getAmount", "getPerson"), relating to the amount of the invoice and the person's name on the invoice. We have a VB program which wants to use these services.

The VB code which would retrieve the amount and person for invoice 10320 is:

```
DIM siena as object
Set siena = createObject("SienaClient")
siena.useService("Invoice")
amount = siena.getAmount(10320)
person = siena.getPerson(10320)
```

Now we would probably want a service which retrieved both the amount and the person together, rather than making 2 calls, so we'd have a service method `getPersonAndAmount`, say:

```
res = siena.getPersonAndAmount(10320)
```

where `res` is now a variant array, the first item of which could be the person and the second the amount. However, as we shall see, the structure of returned values is completely under the service developer's control – so it may be appropriate to have a service which accepts a list of invoice numbers and returns a corresponding list of (person, amount) pairs.

Remember, we are running the VB program on a desktop, and getting the results from a Siena Server somewhere else; but the application code is indeed easy to understand and, given a service and its collection of methods, it's easy to see how one would make a service method call.

Before turning to the server-side implementation of the method, one question often arises, which can be illustrated by the extension of the above example. Suppose that the Purchase and Sales departments both have services, and they each have a service called Invoice (relating to Purchase Invoices and Sales Invoices). How does Siena resolve this?

Simply and intuitively – as you might expect. You would probably define the services not simply as 'Invoice' but in a hierarchy: Purchasing has a set

of services and Sales has a set of services, so we'd expect to change one line of the above code to:

```
siena.useService("Purchasing.Invoice")  
or
```

```
siena.useService("Sales.Invoice")
```

Now to the server side. A large amount of credit for the design success of developing and deploying services must go to Python, the well-established language in which services (and Siena) are written. So the server side code for the Sales Invoice service would be as follows. In a file called Sales.py, you would have:

```
from Siena.Server.API import Service  
  
class Invoice(Service.Service):  
    public_methods = ('getAmount', 'getPersonAndAmount')  
  
    def getAmount(self, invoice):  
        SQL = 'SELECT AMOUNT FROM SALES_INVOICES WHERE INVOICE_NO=%s'  
        % invoice  
        return self.askdb(SQL) [0]  
  
    def getPersonAndAmount(self, invoice):  
        SQL = 'SELECT PERSON, AMOUNT FROM SALES_INVOICES WHERE  
        INVOICE_NO = %s' % invoice  
        return self.askdb(SQL) # return (PERSON,ACCOUNT) as a list  
  
    def askdb(self, SQL):  
        conn=self.connection('SALES_DATABASE')  
        acursor = conn.cursor()  
        acursor.execute(SQL)  
        res = acursor.fetchall()  
        return res[0]
```

And that is all. Let me take you through this a line at a time.

```
1. from Siena.Server.API import Service
```

This imports the module Service.py which enables the Siena "magic".

```
2. class Invoice(Service.Service):
```

This defines the service ("Sales.Invoice" – as this code is held in a file Sales.py) as a Python class, which inherits from the class Service.Service (i.e. a class called Service in the Siena module Service.py).

```
3. public_methods = ('getAmount', 'getPersonAndAmount')
```

This indicates that there are 2 callable service methods.

```
4. def getAmount(self, invoice):
```

This defines the first service method, which takes an invoice number, uses it to construct a SQL statement and return the answer.

```
5. SQL = 'SELECT AMOUNT FROM SALES_INVOICES WHERE INVOICE_NO = %s'
   % invoice
```

The SQL assumes there's a SALES\_INVOICES table with columns AMOUNT and INVOICE\_NO. This line substitutes the value of invoice, as a string, in the placeholder %s.

```
6. return self.askdb(SQL)[0]
```

This returns the required value, by calling the (private to Siena) method askdb(SQL). To explain the [0], as we shall see, askdb() in general returns a list, and we only want the first value from the first entry in the list. This construct gives this element.

```
7. def getPersonAndAmount(self, invoice):
    SQL = 'SELECT PERSON, AMOUNT FROM SALESINVOICES WHERE
    INVOICE_NO = %s' % invoice
    return self.askdb(SQL) # return (PERSON,ACCOUNT) as a list
```

This service method is similar to getAmount(), except that it returns a pair of related values in a list.

```
8. def askdb(self, SQL):
```

Finally, we have the definition of askdb()

```
9. conn = self.connection('SALES_DATABASE')
```

This assigns one of the SALES\_DATABASES pooled database connections to a variable we can use

```
10. acursor = conn.cursor()
```

This establishes a cursor on that database, so that we can do queries, etc.

```
11. acursor.execute(SQL)
```

This runs the query.

```
12. res = acursor.fetchall()
```

This returns the rows from the query, as a list of rows, each element of the list itself containing a list of elements which are the field entries for that row.

```
13. return res[0]
```

This returns the first row as a list of fields.

All the above is python syntax, a fuller explanation of which can be found in the many Python publications available, in book, e-book and web form.

So it is clear how further services within Sales could be added to this file Sales.py, by merely adding:

```
class MyNewService(Service.Service):  
  
    public_methods=("mynewmethod",)  
  
    #(Note there should always be a , in public_methods, even if  
    there is only one entry)  
  
    def mynewmethod(self, args):  
  
        ...  
  
        return something
```

**Note:**  
More detail about writing services is provided in chapter 4.

---

## 3. Client side set up

---

### 3.1 Configuration

The Siena client is configured with a .ini file, named [siena.ini](#), which must be in the same directory as the `sienaclient` executable. The entries are:

```
[Siena]
ServerURI =
# e.g. ServerURI = http://machine.example.com
```

The address of the Siena server, as an http URI. There is no default. This value must be supplied.

```
Timeout
# Timeout = 5
```

Timeout in seconds for initial server contact. If the first call to the server takes longer than this, the operation is aborted and an error is returned to the client. Default is 0, meaning no timeout. The client will wait forever if the server does not respond.

---

### 3.2 Licensing

Licensing information is held within the file [siena.lic](#), which must also be the same directory as the `sienaclient` executable.

```
[License]
Key = 2002-07-29:example:f4ea1d4261ccc2660579f953c69b9898
```

If the license key is invalid or has expired, the COM component and dashboard will refuse to run.

---

### 3.3 Using the Siena COM interface

Siena registers a COM dynamic server with the name 'SienaClient'. You create an instance of it with, e.g. (Visual Basic):

```
Dim osiena As Object
Set osiena = CreateObject "SienaClient"
```

Siena service methods are arranged into a naming hierarchy, of the form `domain.subdomain.service.method`, with any number of sub-domains. An example is `ipm.control.liststarted`. Conceptually, the code to invoke a service method looks like:

```
result = osiena.ipm.control.liststarted()
```

but COM does not support such constructs. Instead, the service must be selected before its methods can be called:

```
osiena.useservice "ipm.control"  
result = osiena.liststarted()
```

Once the service has been selected, it is remembered until a different service is selected. Several methods within a single service can be called without having to repeatedly call `useservice`.

To use different servers, such as the local server, use the `useserver` method:

```
osiena.useserver "Local"  
result = osiena.someservice(...)  
osiena.useserver "Siena"
```

Use the "Siena" server to select the default server.

---

### 3.3.1 The useformat method

The `useformat` method allows the client to specify easily the format that lists (and lists of lists) will be passed to the server, and the format the client would like to receive any lists. This is only required for advanced users accessing the COM client-side stub interface to Siena.

It is common for service developers to return list of strings or integers to the calling program. In most languages, there is a list or vector structure which is familiar and convenient to use for 1 or more dimensions (lists of lists, for example). It is the nature of Microsoft COM, that a list appears in the calling program as a variant array. (as an extension of the design that a single result is returned as a variant). For some applications, and application developers, this mechanism is clumsy; for example, a VB coder may wish to send a list of keys as a string which is comma delimited, but to tell Siena that this should be handled as a list. Likewise, it might be more convenient to receive a list as a return value as a, say tilde-delimited string. Clearly such structures can always be represented in XML as a string, and there is no requirement for multi-dimensional lists in this case, the client program and service merely passing a string and then each decoding and encoding the XML as it is seen fit.

Calling useformat on the siena client instance enables the calling program to determine at runtime which format it should be sent and received in. The syntax of the useformat is as follows:

```
useformat("send_format", "recd_list_format", "recd_lists_format")
```

To illustrate this, if there is a service called `getReverse` which takes a list and reverses it, and the client wants to send a string with comma delimiters, the following code would be appropriate (in Delphi):

```
siena.useformat(',', '~', '');  
myresult := siena.getReverse ('arg1,arg2,arg3');
```

The service call results in a reversed list, and as specified by the client, the delimiter is '~', so `myresult` is returned with the value 'arg3~arg2~arg1'.

Without the useformat call, the VB developer would have to construct a variant array, each element of which being "argn" in this case, and would have to loop through the resulting variant array.

It is also a limitation of COM that multi-dimensional lists cannot be successfully passed (except in the special, and not often useful, case of a 2 dimensional "list of lists" where all it is necessary that the length of all lists in the second dimension are equal). So to handle the multi-dimensional case, the useformat service call allows the application developer to determine the delimiter convention to be used in calling and receiving such multi-dimensional lists from services. The third argument of the useformat function determines the format the list of lists is received in, and the first argument again specifies the format for sending the lists.

Expanding our imaginary `getReverse` function to handle lists of lists, it will now reverse the entries within each list, and the order of the lists themselves.

```
siena.useformat('#', '|', '|@');  
myResult := siena.getReverse2('a,b,c#1,2,3');
```

By the format specified in useformat, the value of `myResult` is now set to '3|2|1@c|b|a'.

---

## 3.4 Language specific details

There now follow brief introductions to client side use of Siena from the environments of VB, Delphi and Java.

---

### 3.4.1 Visual Basic Clients

As introduced above, the `siena` object must be created. It is recommended (but not essential) that this is done when the form is loaded. This is because it can take a few seconds, and it is desirable to have this wait only once, rather than each time the object is used in making service calls. This is done by declaring the object publicly outside any methods, and then creating the `SienaClient` object in the form load method. Example VB code follows.

```
Public siena As Object

Private Sub Form_Load()
    Set siena = CreateObject("SienaClient")
End Sub

Private Sub Command1_Click()
    siena.useservice "sienademo.serv"
    Response.Text = siena.test
End Sub
```

---

### 3.4.2 Delphi Clients

The `siena` method calls look the same in delphi as in VB (above). Setting up the module is slightly different. The following two entries need to be added to the 'uses' section of the module:

```
uses
    ComCtrls, comobj;
```

The instantiation of the OLE object cannot happen until the variable is declared. In the private declarations section of the module, write

```
private
    siena : Variant;
```

Of course, this variable can be named anything. `siena` is used here to be consistent.

As recommended above in the VB example, the instantiation of the OLE object can be done in the form create method. Within the `implementation` section of the module, a simple procedure is needed:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    siena := CreateOleObject('SienaClient');
end;
```

Now the delphi module is ready for custom service calls. The code for this will be the same for all clients, since it is server and service dependent:

```
procedure TForm1.Command1Click(Sender: TObject);
begin
    siena.useservice('sienademo.serv');
    Response.Text := siena.test;
```

end;

---

## 3.5 Developer Dashboard

The Siena Dashboard presents trace messages resulting from problems in contacting the Siena server, or errors in the service implementations. To start the Dashboard, double-click on [Dash.exe](#) in the Siena client distribution. The Dashboard installs itself in the Windows system tray; double-click the Dashboard icon (a script "S") to open the Dashboard window. The window opens automatically when trace messages are written to it.

The main part of the window shows the log messages that have been output.

The [Debug mode](#) checkbox determines whether log messages are output or not; unchecking it suppresses log messages. There is normally no need to uncheck it.

The [Clear log](#) button simply clears the contents of the log window.

The choice box allows you to choose between the main Siena server, the local server, and any other servers you may have configured in [siena.ini](#).

The [Retrieve log](#) button fetches the end of the log file from the Siena server selected in the choice box. This allows you to see details of exceptions thrown by services, for example.

The [Reload server services](#) button asks the Siena server selected in the choice box to reload the service implementations, to pick up modifications or additions. It will *only* succeed if the Siena server is in debug mode. In deployment mode, service reloading is disabled for security reasons.

In the 'File' menu, the 'Save log' option allows you to save the current contents of the log window.

---

## 4. Server Side Setup

This section explains how to set up the Siena Server.

---

### 4.1 Team and Personal Server Configuration

The difference between the Team and Personal Siena servers is accessibility. Personal servers can only be accessed by the client on the machine where the server is installed, whereas Team servers are accessible from multiple clients. Configuration for both servers is the same. The file [Config/system.ini](#) contains two sections.

```
[Database]
#otherdb1 = iODBC:DSN=datasourcename;UID=userid;PWD=password
#otherdb2 = unixODBC:DSN=datasourcename;UID=userid;PWD=password
```

This sets up communication between the Siena server and databases.

```
[LocalServer]
# Port = 1234
```

This does not have to be specified. If unspecified, the default port is 8000.

---

### 4.2 Enterprise Server Configuration

The Enterprise Server uses a standard [Config/system.ini](#) file for setup.

The Siena Enterprise Server will have to know a few things about your server setup, the path locations, names and a few other things. This information is all stored in the Siena Enterprise Server setup file [system.ini](#) found in the [Config/](#) subdirectory of the Siena installation.

Here is a sample [system.ini](#) file with annotations for the different parameters (you can safely leave unannotated parts unchanged):

```
#####
# Siena Enterprise Server Configuration File
#
# This file has optimized settings for use of the Siena Enterprise
# Server as a stand-alone server.
#
```

```

...
### Site configuration #####
#

[Site]
#
# Site's literal name

# name = Siena Enterprise Server Test Site
      Enter the site's literal name here.

#
# Produce debug output ? This should be on (1) for development
# installations and off (0) for production sites.

# debug = 0

```

*Development* installations of the servers can only have debug = 1. Changing this flag will not change the system.

Enabling debug output will produce a verbose Python traceback listing of the error location and send it to the browser instead of the formatted page in case of an error situation to help the site designer in debugging dynamic web pages. This listing may include information which you do not want to be published.

With debug switched off, only a short message in SGML comments (<!-- ... -->) will be inserted into the page. The message will not include internal information and won't be visible in the browser.

```

#
# Disable the use of cookie ID for user-tracking ? The system will
# then try to use URL pathinfo IDs to do the tracking. This is less
# likely to work though, since there situations where the system
# cannot detect a URL in the page output (e.g. in JavaScript) and
# these links then cause the tracking to break. Default is to
# prefer cookie-based tracking over pathinfo tracking.
#

disable_cookie_ids = 0

```

Since Siena does not need to use Netscape cookies for client tracking, these can safely be switched off.

Note that you cannot switch off the handling of the special HTTP header `x-eas-visitorid` which Siena uses for client tracking, provided the client application interface supports this.

```

[Database]
#
# This section defines the database connections to be maintained by
# the Siena Enterprise Server. Connections are established through

```

```

# the attached ODBC manager (iODBC or unixODBC on Unix, Windows ODBC
# manager on Windows).
#
# Each connection given here is opened at system startup
# and kept alive during the processing lifetime of the
# different parts of the system. Since the server uses a
# pool of worker processes, this means that the databases
# must be able to handle up to enough connections to
# accomodate for the traffic expected at the site.
#
# The format for each connection is as follows:
#
#   <connectionname> = <method>:<data source logon string>
#
# where <method> is one of iODBC, unixODBC or Windows depending on
# which ODBC managers are available on your system (the Siena
# supports the following ODBC managers: iODBC, unixODBC and
# the Windows ODBC managers), e.g.
#
#   myconnection = unixODBC:DSN=test;UID=test;PWD=test
#
# The database connection will be made available in the Siena
# Application Server under the given connection name.

```

**Note that the default connection method is to use the iODBC manager, since this is the preferred way of connecting to a database.**

```

# The logon string has to use the standard ODBC DSN notation, i.e.
#
#   DSN=<data source name>;UID=<user id>;PWD=<password>[;<arg>=<value>]
#
# <data source name> has to be the data source name as defined
# in the ODBC manager configuration section for the database
# backend (see ~/siena/.odbc.ini for details).
#
# The arguments given after the password are optional and
# depend on the ODBC driver used for the connection. Please
# consult you ODBC driver manual for information on the
# available switches.
#
#
# The connection with name "system" will be used by the system to
# store site information.
#
# The default connection string used by the system is given below. If
# your parameters differ, uncomment the line (remove the '#') and
# adjust the settings to your needs.
#
# Default (see ~/.odbc.ini for details):
#system = iODBC:DSN=sienadb;UID=siena;PWD=sienapw
#
# SAP DB backend:
system = iODBC:DSN=localhost:SIENADB;UID=siena;PWD=sienapw

```

**The Siena Enterprise Server is optimized to work with the SAP DB backend, so you are encouraged to use this option.**

```

# MySQL backend:

```

```
#system = iODBC:DSN=mysql;UID=siena;PWD=sienapw
```

```
#  
# The connection with name "systemlocks" will be used by the system to  
# maintain row level locks. It is used in auto-commit mode.  
#  
# The default connection string used by the system is given below. If  
# your parameters differ, uncomment the line (remove the '#') and  
# adjust the settings to your needs.  
#
```

The `systemlocks` setting should use the same value as the `system` connection name (the `TRANSACTION=0` addition is not necessarily needed – the system will automatically switch off transactions on this connection).

```
# Default (see ~/.odbc.ini for details):  
#systemlocks = iODBC:DSN=sienadb;UID=siena;PWD=sienapw
```

```
# SAP DB backend:  
systemlocks = iODBC:DSN=localhost:SIENADB;UID=siena;PWD=sienapw
```

The Siena Enterprise Server is optimized to work with the SAP DB backend, so you are encouraged to use this option.

```
# MySQL backend:  
#systemlocks = iODBC:DSN=mysql;UID=siena;PWD=sienapw
```

```
#  
# Example for connecting to other external databases known to the  
# ODBC manager.  
#  
# You can connect the system to any number of databases by adding  
# more connection definitions to this section. The ones given here  
# are only examples; they are not defined per default.  
#
```

```
#otherdb1 = iODBC:DSN=datasourcename;UID=userid;PWD=password  
#otherdb2 = iODBC:DSN=datasourcename;UID=userid;PWD=password  
#otherdb3 = iODBC:DSN=datasourcename;UID=userid;PWD=password  
#otherdb4 = unixODBC:DSN=datasourcename;UID=userid;PWD=password  
#otherdb5 = unixODBC:DSN=datasourcename;UID=userid;PWD=password  
#otherdb6 = unixODBC:DSN=datasourcename;UID=userid;PWD=password
```

Here you configure any other database connections you may want to use.

As you can see, it is readily possible to use different methods of connecting to a database within one server.

Some ODBC drivers work better with iODBC, others with unixODBC, e.g. the EasySoft Oracle driver does not support iODBC at all, so unixODBC is the only option here.

```
[Services]
```

```
#  
# This section enables configuration of important aspects of the optional  
# and user supplied service modules. All properties will use the type  
# string.  
#
```

```

#
# If turned on, services are run in debug mode which makes them
# more verbose in case of errors. On production systems this should
# be turned off to avoid security risks due to too much information
# being passed back to the client. During development this will
# return complete tracebacks to the client in case an unexpected
# exception occurs in a service call.

# debug = 0

```

Setting `debug` to 1 is useful for development system as it provides the developer with more information. It should not be enabled on production systems, since that would publish too much internal information and could introduce subtle security risks.

```

[Licenses]

#
# Licensee's name.
#
# licensee = Licensee name
licensee =

[System]
#####
#
# There is usually nothing serviceable below this line.
#

# Disabling hit recording results in a performance boost,
# so leave at 0
record_hits = 0

# Timeout for Visitor records (days:hours:minutes:seconds)
visitor_timeout = 00:18:00:00.00

```

Visitor records are used by Siena to store client application session data.

The data should never persist across client application invocations, so we lower the timeout to 18 hours.

Visitor timeout refer to the time between two requests using the same Visitor ID. If the time between two requests is longer than the period indicated here, a new Visitor ID is generated by Siena and the old ID times out. The next run of the Siena garbage collector will then remove the associated Visitor records from the system database.

```

# Interval between automatic garbage collection runs in seconds
gc_interval = 32200

```

To avoid having too much load on the system database due to thousands of unused Visitor records, it is a good idea to run the garbage collector more often.

The above value will let it run twice a day.

```
# Enable usage of the DirectoryControl which maintains an
# internal sitemap of the site objects. Possible values are
# 0 (disable) or 1 (enable). Leave at 0.
enable_directory = 0
```

This variable is unused, and must be set to 0.

```
# Value for the System View's visitor_views attribute;
# leave at "Engine".
default_visitor_views = Engine
```

Siena comes with a pre-configured set of objects which implement the standard URL-to-service mappings documented in this guide.

The above setting enables this set of objects per default. Disabling it will cause the URL-to-service mappings to stop working.

**Note:**

Settings which are not mentioned here or are left undocumented should not be altered.

---

## 4.3 Installation of User Services

Installation of user written service modules is easy. The next few sections describe the typical installation procedure.

---

### 4.3.1 The Lib/UserServices/ Directory

User defined services can be installed by copying the Python modules and/or package directories into the [Lib/UserServices/](#) directory in the Siena installation directory.

This directory is configured as a Python package (i.e. it includes a `__init__.py` file) and scanned at server startup time for available services to be registered and published by Siena automatically. Details on how the module and package names map to actual service names are described in section 5.2.

**Note:**

Do not remove the `__init__.py` module file from the directory – the server won't start up without this file or if the file is not readable by the *siena* user.

---

### 4.3.2 Managing Service Modules Access Permissions

The only requirement for Siena to find and register the services is that the module files are readable by the *siena* user account under which the server executes.

Service developers can use the Unix file permission logic for managing file level access to the various parts of the directory structure underneath `Lib/UserServices/` in order to control access to certain parts of the structure by other service developers.

Example:

Suppose the services administrator wants to restrict write access to services for division “food” of a company to the Unix user group *fooddiv*.

To do so, the services administrator would set the ownership of the service files in e.g. `Lib/UserServices/com/company/fooddiv` to *siena.fooddiv* and then disallow write access to all others (including the user *siena* which should never be member of this group), i.e.

```
chown -R siena.fooddiv Lib/UserServices/com/company/div1
chmod 460 Lib/UserServices/com/company/div1/*
```

---

### 4.3.3 Updating the Service Registry

Once a new service is installed in the `Lib/UserServices/` directory, Siena needs to be notified of this change. There are three ways to achieve this:

1. initiate a service module reload (as *root*: `su - siena -c "ServerServices --reload"`)
2. call the Service `server.system.reloadServices()` e.g. from the client (only possible if the Enterprise Server is running in service debug mode)
3. reset the Enterprise Server (as *root*: `su - siena -c ResetServer`)

The server will then rescan the `Lib/UserServices/` directory and automatically update the Service registry.

This reload process can take a few seconds to complete.

Solutions 1 and 2 have the advantage of keeping the server online during the reload procedure. The downside of using these is that they may leak memory, since it cannot be assured that the old service modules are removed from memory by the reload.

---

## 5. Writing Services

Siena uses the Open-Source programming language Python (<http://www.python.org/>) as service scripting language.

Consequently, writing services for use in Siena is just as easy as writing a Python class exposing a set of methods.

All the complicated work like maintaining database connections, handling persistent session data and implementing protocol stacks is handled by Siena.

The next few sections will give an overview of how to write Siena services.

**Note:**

Knowledge about writing Python classes and methods is needed in order to understand the various aspects involved in authoring services with Siena.

---

### 5.1 Concepts

The design of Siena focuses on two simple ideas: writing services should be easy, and publishing services should be simple.

---

#### 5.1.1 Writing Services should be easy

While there are various ways to write and run Services today, most of these focus around the Java programming language and require running and maintaining complex server setups such as the J2EE-compatible application servers IBM WebSphere or BEA WebLogic to publish these to client applications.

This may be the right solution for complex services such as online flight booking services, but introduces a lot of administrative and development overhead for everyday, small to mid-sized projects.

Siena focuses on ease of development and deployment rather than trying to compete with full-blown J2EE server environments – even though it does provide much of the J2EE functionality in its Python environment.

To achieve this, Siena makes use of the Python programming language and drop-in automatic configuration to increase development productivity to a level several times higher than that which can be achieved in a typical J2EE development environment.

Writing services becomes just as easy as writing a Python class with a few methods – database connectivity and session management, automatic recovery from programming errors and many other typical service authoring tasks are dealt with in the server rather than putting the load on the service designer.

Once debugged and tested, deployment of newly written services is then just a matter of copying a set of service modules to the server. Siena will automatically register them after a service reload and take care of keeping them online.

Another advantage, which Python as development language shares with Java, is that you can write Python programs which run on more than just one platform.

Python uses the same strategy as Java: it compiles the program code to platform independent Python Byte Code. So whether you are developing on Windows, Macintosh or Unix is up to you. The resulting code can easily be ported to the server – in most cases without any change.

---

### 5.1.2 Publishing Services should be as simple as possible

As mentioned in the introduction, there are several different ways to have client applications talk to a service engine.

The available architectures include rather simple method invocation protocols encapsulated in HTTP such as XML-RPC and more complex ones which support extending the range of supported data object, e.g. SOAP.

There are also a number of different communication techniques such as various inter-process communication (IPC) solutions which allow setting up single tier setups and complex object brokers such as CORBA which use their own protocol stacks.

Traditionally, each architecture requires having to rewrite the service in a slightly different way, introducing code duplication, maintenance issues and reducing develop productivity in general.

In an ideal setting, services would only be written once and then automatically made available to as many different architectures as needed.

Siena aims at this ideal by providing a single, consistent interface for the service programmer to develop against. This interface then allows the server to publish the services and methods in many different ways, without having to change single line of code in the service implementations.

---

## 5.2 Naming Conventions

Siena is designed to support drop-in configuration of services. In order to make this as simple as possible, a special naming scheme was chosen which all service implementations have to comply to in order to get published through Siena.

---

### 5.2.1 Naming Scheme

Siena service method names are made up of two components: *service domain* and *method*. The *service domain* is made up of, at minimum, the class name of the service. If the service is contained in a sub-directory of [Lib/UserServices/](#), the *service domain* must follow this structure, thus uniquely identifying each module. This will become clear in the examples that follow.

Names may contain alphanumeric characters and the underscore. Spaces and other special characters are not allowed.

Examples:

- `MyService.myMethod`
- `com.company.division.getDatabaseRecord`
- `com.company.getDivisionNames`

---

### 5.2.2 Special Service Namespace

One service domain is reserved for internal use by the system:

- `server.*` refers to server side services and methods which map to special service package on the server,

e.g. `server.system.reloadServices()` could be a method to have the server reload the registered services.

---

### 5.2.3 Service and Method Name Lookup

Service name lookup is done in two steps:

1. case-sensitive and then
2. case-insensitive, if the first pass should fail.

This allows implementing case-sensitive service and method names which is important for e.g. SOAP and WSDL while retaining the possibility to also address them in case-insensitive ways for those client applications which normally run in case-insensitive mode, e.g. Visual Basic, COM, etc.

Method names are looked up using a similar scheme:

1. case-sensitive in the service class' `.public_methods` attribute and then
2. case-insensitive in the `.public_methods` attribute from left to right, if the first pass should fail.

---

### 5.2.4 Service Name Scopes

The central concept of naming services within Siena is that of using two different scopes:

- *internal names* and
- *external names*.

Internal names are the names under which the services are registered within Siena, while external names are the names under which a client application can call the published services.

#### Internal Service Names

Internal names directly map to the full module path name of the module defining `Service` subclass combined with the class name using a single dot ("`.`"),

e.g. `packagename.modulename.classname`.

Services are managed in two separate Python packages:

- one is called `UserServices` and can be modified by the service programmer,
- the other is `Siena.Server.Services` which is reserved for server APIs and exposed under the `server.*` service domain (e.g. `server.system.getServices`) and cannot be modified by the programmer.

Services resides in the `Lib/UserServices/` directory, or sub-directories thereof.

#### Example:

##### File `Lib/UserServices/MyDomain.py`:

```
from Siena.Server.API import Service
class MyService(Service.Service):
    pass
```

Since this file resides in the `UserServices` package on the server, the complete internal name of this service is `UserServices.MyDomain.MyService`.

### External Service Names

External names are the names under which a client application can call a Siena service implementation.

### Mapping External to Internal Names

The actual mapping of the incoming request using the external name to the internal name is straightforward:

The standard procedure for the mapping is as follows:

- any incoming request to a service in the service hierarchy `server.*` is mapped to the internal name `Siena.Server.Services.*` (the string `"server."` is replaced with `"Siena.Server.Services."`)
- all other requests are mapped to `UserServices.*` (the string `"UserServices."` is pre-pended to the external name)

As a result, the external service domain `Something.MyService` would be mapped to the internal name `UserServices.Something.MyService`, while `server.system` would map to `Siena.Server.Services.system`.

---

### 5.2.5 Service Parameter Names

Siena can determine the name of the parameters a service method defines by looking at the Python method implementation. This makes it possible to use both named parameter references (e.g. `method(name='Joe')`) as well as anonymous order based ones (e.g. `method('Joe')`).

Note that this only works for service methods which are implemented in Python. If you add and expose methods which are written in C, then this mechanism will only work for those methods which were written to expect keyword parameters. Please consult the Python C API reference on how this is done.

---

## 5.3 Organizing Service Implementations

Services in Siena are defined as subclasses of the `Service` class which is defined and exposed in the `Siena.Server.API.Service` Python module.

To write a service, you just have to import the Python module in your service module and then subclass from it:

```
from Siena.Server.API.Service import Service
class MyService(Service):
    pass
```

Dropping this module into the services directory will then make the service available to client applications (after a server service reload). The next section details this process.

---

### 5.3.1 Services Directory and Auto-Registration of Services

To register a `Service` subclass, the defining module has to reside in a special Python package directory of the Siena installation: `Lib/UserServices/`, called the *Services Directory*.

All modules and packages (directories having a file `package/__init__.py`) in this directory are scanned recursively at server startup time for subclasses of the `Siena.Server.API.Service` class. Such classes are then automatically registered to work as services.

Only those subdirectories of the `Lib/UserServices/` directory are recursively scanned which have a Python module called `__init__.py`. This is how Python identifies package directories. The module file can be an empty file, but does have to exist in order for the directory to be scanned by Siena.

You can also have sub-package directories within a package directory. It is possible, and often useful, to setup a directory structure [Lib/UserServices/company/division/project/](#) to manage multiple projects of a company division. The only thing to remember is that each level of this structure must have an `__init__.py` module in place which marks the directory as Python package.

An important feature of this logical mapping is that it can be put to use to manage service module write permissions: since the modules live in the file system, the server's file permission system allows managing access to the code. See section 4.3.2 for details on this feature.

#### Example:

##### [File Lib/UserServices/MyDomain.py:](#)

```
from Siena.Server.API import Service

class MyService(Service.Service):

    # Tuple of published methods:
    public_methods = ('mul2',)

    # Service methods:
    def mul2(self, x):
        return 2*x
```

This service would get registered under the internal name `UserService.MyDomain.MyService`. The method's external name is `MyDomain.MyService.mul2`.

More details about the used mapping of module, class and method names to public names and which methods are considered public are described in the section 5.2.

Instructions on how to update the service registry at server run-time and how to manage service module access on the server are explained in section 4.3.

---

### 5.3.2 Registration Error Handling

Siena will try to import all modules and packages found in the [Lib/UserServices/](#) package directory at server startup time.

If an import should fail due to an `ImportError` or `SyntaxError`, these errors will be written to the server's log file in [System/system.log](#) and processing will continue, but the erroneous service implementations won't be available for clients to call.

**Note:**

You *should not* include complicated startup code in the top-level of the service modules or packages, since this code will execute very early during server startup at service module import time and might cause a deadlock situation which results in the server not completing its startup.

Examples of such situations are:

- a module which tries to communicate with the server even though it hasn't completely started yet,
- a circular import situation which cannot be resolved by the server,
- a long running setup procedure which takes more than the configured maximum busy time for a service (default is two minutes).

To test for this situation, temporarily move all service modules and packages out of the [Lib/UserServices/](#) package directory and try to restart the server. If it comes up without problems, then you should check all recent service module additions for errors and bugs before reinstalling them.

---

### 5.3.3 Importing Python Modules

If the user service [MyService.py](#) requires the use of another python module, the module [MyService.py](#) will include the line

```
import PythonFile.py
```

The isolated file [PythonFile.py](#) needs to simply be placed in the [Lib/UserServices/](#) directory along with [MyService.py](#).

---

## 5.4 Service Implementations

At service registration time, the internal name of a service class is determined by following the rules for internal names described in section 5.2.4. There are two kinds of services which Siena provides to the service author:

1. Standard Services
2. Domain Services

The only difference between the two is the way in which their methods can be addressed and this is achieved by using different base-classes for their definitions.

---

#### 5.4.1 Standard Services

Standard services are all service classes which have the Python class `Siena.Server.API.Service.Service` as base-class.

Their methods can only be addressed using the class name as service name, e.g. the method `MyMethod` of class `MyService` in the module `Something.py` would be available under the external name `Something.MyService.MyMethod`:

File `Lib/UserService/Something.py`:

```
from Siena.Server.API.Service import Service
class MyService(Service):
    def MyMethod(self):
        return 'hello world !'
```

---

## 6. Domain Services

A domain service is one in which the service class name is identical to the name of the module in which it resides. To define domain methods, a service class must derive from `Siena.Server.API.Service.DomainService`.

Siena will then map the dotted *module name* to this service class, allowing method calls on the service domain.

The name of the class is not important, `Domain` or the module name should be used per convention and only one such class may be defined per module:

File `Lib/UserService/MyDomain.py`:

```
from Siena.Server.API.Service import DomainService
class MyDomain(DomainService):
    def test(self):
        return 'hello world !'
```

Here, the method `test` would become available under the external name `MyDomain.test`.

---

### 6.1.1 Reserved Service and Method Names

When writing a service as subclass of the `Service` or `DomainService` class, you have to be aware of a few reserved symbols which you should not use in your implementation.

These symbols are documented in section 0 of this guide.

---

### 6.1.2 Examples

For the purpose of these examples, let's assume that all modules reside in or under the `Lib/UserServices/` directory of the server installation.

## Domain methods: domain.method

File `domain.py`:

```
from Siena.Server.API.Service import DomainService
class domain(DomainService):
    public_methods = ('method', )
    def method(self):
        return 'hello world!'
```

## Service methods: domain.service.method

File `domain.py`:

```
from Siena.Server.API.Service import Service
class service(Service):
    public_methods = ('method', )
    def method(self):
        return 'hello world!'
```

## Sub-domain method: domain.subdomain.method

File `domain/__init__.py`:

```
# This directory is a Python package
```

File `domain/subdomain.py`:

```
from Siena.Server.API.Service import DomainService
class domain(DomainService):
    public_methods = ('method', )
    def method(self):
        return 'hello world!'
```

## Sub-domain service method: domain.subdomain.service.method

File `domain/__init__.py`:

```
# This directory is a Python package
```

File `domain/subdomain.py`:

```
from Siena.Server.API.Service import Service
class service(Service):
    public_methods = ('method', )
    def method(self):
        return 'hello world!'
```

---

## 7. Siena options

---

### 7.1 Server synchronisation

Siena Enterprise Servers can be optionally configured as Distributed Hub Servers. When operating in this mode, such servers are designed to synchronise Client Hub Servers when requested to do so. Typically this configuration exists when an organisation has an Enterprise Server and several Team and Personal Servers, each of these Servers performing a "local" service to applications on the desktop or group computer on which they reside. Such local services would be used, for example, where an organisation wishes to have some of its Business Logic running on the processors of the client desktops, rather than having them all unnecessarily share the processing capability of a central server, together with the related overhead of network calls.

In these configurations, the Enterprise Server has a file [UserComponents/LocalServer.cfg](#) which contains sections called "applications". Now, the organisation might have an application called "Cheeky" and the Cheeky application requires, say, a service module called [UsefulServices.py](#) to be up-to-date on the Client Hub. For this, [LocalServer.cfg](#) uses the path to the file relative to the Siena directory on both the client and server, which by default have the same directory structures:

```
[Cheeky]
ApplicationServer/Lib/UserServices/UsefulServices.py
```

An alternative syntax is permitted:

```
[Cheeky]
filename_on_client = filename_on_server
```

where the (full) filename includes the path from the respective Siena directories, or alternatively absolute pathnames can be used for either filename is acceptable. Note that there is no restriction on the files that may be mapped or uploaded, so the Server Hub may be used creatively to upload not only Siena services but .ini or .cfg files – in fact any file.

On the Siena Servers acting as Client Hubs, there is a simple entry in the configuration file [siena.ini](#). Under the section whose name is a Hub Server, there is an entry which takes the form

```
[Siena]
ServerURI = http://machine.example.com
```

```
Timeout = 5
```

This section was documented in chapter 3. The new line in the above example would be

```
application = Cheeky
```

If more than one application's files are being synchronised, the syntax is

```
application = appl,app2,...
```

The following additional section to [siena.ini](#) must also be added,

```
[Local]  
ServerURI = http://localhost:8000  
Sync = Siena
```

to indicate the URI of the local server, and that it is synchronised with the main Siena server.

---

## 8. Service API Reference

This section defines the server application programming interface (API) provided by Siena.

---

### 8.1 “Siena.Server.API.Service” Module

This module provides all the necessary Python base-classes to write services with Siena. To import it in a service module, use:

```
from Siena.Server.API import Service
```

If you just want to import a few symbols, such as the service base-classes from the module, write:

```
from Siena.Server.API.Service import Service, DomainService
```

The available symbols are described in the next few sections:

---

#### Class "Error"

Base class for service related errors.

```
Base class(es): StandardError
```

---

#### Class "CancelError"

Error raised in case a service request could not be processed.

Transactions are rolled back by this exception, if it is not caught in service processing code before getting back to the Request object processing the request.

```
Base class(es): ServiceCancelError, Error
```

---

### Class "AccessForbiddenError"

Error raised in case a service request was not permitted to continue due to access restrictions.

Transactions are rolled back by this exception, if it is not caught in service processing code before getting back to the Request object processing the request.

```
Base class(es): ServiceAccessForbiddenError, Error
```

---

### Class "Service"

Base class for writing services.

#### **Attributes:**

```
.public_methods = ()
```

This attribute has to be set to a tuple of strings defining the names of the methods of the `Service` subclass which should be published through the various service interfaces. Methods not listed here are not available as services.

#### **Methods:**

```
.check_access(methodname, args, kws)
```

Check access to the method referenced by `methodname` and return 1 (access granted) or 0 (access denied).

Request objects call this method prior to executing the method with the given arguments to check whether access should be given or not. The `methodname` passed in is the internal name of the method, i.e. `method.__name__`, in that case.

`.check_access()` is only called for registered public methods (i.e. those which are returned by `.get_public_methods()`).

```
.connection(name)
```

Returns the `Connection` object for connection name. See the Appendix A for details on the available methods and attributes of `Connection` objects.

```
.service_external_name()
```

Return the external name of the Service.

The external name of the Service is defined by the calling Request object. As a result, this method only works in the context of an external call to the Service.

```
.service_find_method(methodname)
```

Returns the public method `methodname` as bound method or None in case no such method exists.

`.service_find_method` looks up the method in two phases: first it tries the case-sensitive name, then, if that fails, it compares the `methodname` to all entries in `.public_methods` in a case-insensitive way. The first match is taken to be the correct method.

```
.service_get_method(methodname)
```

Returns the public method `methodname` as bound method.

Raises an `AttributeError` in the case that no such method exists.

```
.service_has_method(methodname)
```

Return 1/0 depending on whether the service provides a public method `methodname` or not.

```
.service_instance(path=(), context=None, **kws)
```

Create an instance of the Service which can then be used to call service methods.

Copies the Service object and sets per-request attributes as needed.

```
.service_internal_name()
```

Return the (internal) name of the Service.

The name is derived from the module and class name of the defining class. It includes the complete dotted path name to defining class.

```
.service_methods()
```

Return a dictionary of all public service methods.

The dictionary maps method names to service describing data objects.

The data objects themselves are not defined, but will hold information about the service methods such as type information, documentation, etc.

```
.service_name()
```

Return the (internal) name of the Service.

The name is derived from the module and class name of the defining class. It includes the complete dotted path name to defining class.

`.service_startup()`

Initialize the service object.

Note that this object will live as long as the servicing process lives. To process a request, the object is first shallow copied by calling `.service_instance()` and the copied version then executes whatever action is needed to fulfill the request.

### Reserved Symbols for this Class

None of these symbols may be used for methods or attributes in subclasses of the `Service` class.

`.service_*`

All symbols starting with "service\_" are reserved for server use.

`.startup`

Needed internally by server.

`._*`

All symbols starting with "\_" are reserved for server use. This includes special Python methods such as `__init__`, `__str__`, etc.

`.__init__`

Needed internally by server. Do not override !

`.public_api`

Needed internally by server.

`.get_api`

Needed internally by server.

`.control_*`

All symbols starting with "control\_" are reserved for server use.

`.reset_control_user`

Needed internally by server.

`.set_control_user`

Needed internally by server.

`.creationdate`  
Needed internally by server.

`.clone`  
Needed internally by server.

`.copy`  
Needed internally by server.

`.process`  
Needed internally by server.

`.finish`  
Needed internally by server.

---

## Class "DomainService"

Special Service subclass which can be used to define a service to implement domain (or module level) methods.

Note that the name of this service is the module defining the service. The class name itself is not included in the name. By convention, the `DomainService` subclass should be called 'Domain' or use the same name as the module.

**Note:**  
There may only be one such class defined per service module.

*Base class(es): Service*

### **Methods:**

`.service_name()`  
Return the name of the DomainService.  
The name is derived from the module name of the module defining the class. The class name itself is not included in the name.

### **Reserved Symbols for this Class**

Same as those of the class `Service`.

---

## 8.2 Data Types

Siena provides simple ways for client programs to access services, using a Siena Client. Depending on the language of the client program, care has to be taken how different data types are handled.

The next sections explain the various pre-defined data types appropriate to the various supported client languages. Internally data types are passed between the Siena client and Siena server using XML-RPC, so the data types must always conform to a valid XML-RPC datatype.

---

### 8.2.1 COM Mapping

The following table lists the COM type to Python type mapping used in Siena. Incoming requests are automatically mapped to the mentioned Python types, while Python return values are automatically encoded using the given COM types.

<i>COM Type</i>	<i>Python Type</i>	<i>Python Built-in Constructors</i>
String	String	<code>str()</code>
Integer	Integer	<code>int()</code>
Float	Float	<code>Float()</code>
Variant Array	List	<code>List()</code>

Combinations of the above types are also allowed, e.g. a sequence of integers, a list of lists, etc.

---

### 8.2.2 Java Mapping

The following table lists the Java type to Python type mapping used in Siena. Incoming requests are automatically mapped to the mentioned Python types, while Python return values are automatically encoded using the given Java types.

<i>Java Type</i>	<i>Python Type</i>	<i>Python Built-in Constructors</i>
String	String	<code>str()</code>
Int	Integer	<code>int()</code>
Float	Float	<code>Float()</code>
Vector	List	<code>List()</code>

Combinations of the above types are also allowed, e.g. a sequence of integers, a list of lists, etc.

---

### 8.2.3 Unhandled Types

There could be situations in which Siena cannot map a return value to a corresponding XML-RPC type, for example by a coding error by a service developer.

In this case, Siena will generate a XML-RPC Fault message having the `XMLRPC_BAD_RESPONSE` fault code and send this back to the client. Any pending database transactions would be rolled back.

---

## 8.3 Exceptions and Fault Codes

Python exceptions raised during the processing of a service request are usually passed back to the Siena Client, which presents them to the client in a standard manner.

---

### 8.3.1 XML-RPC Fault Codes

XML-RPC fault messages are composed of a fault code integer and a fault string which explains the error situation. The following fault codes are used by Siena:

`XMLRPC_DOMAIN_NOT_FOUND = 10`

The requested service domain (Python module or package) was not found.

This can be due to a service module causing import problems, wrong file permissions on the modules files or simply missing service modules.

`XMLRPC_SERVICE_NOT_FOUND = 11`

The requested service (Python class defined in the domain module or package) was not found.

This can be due to a misspelling of the service name, a missing import in the top-level service domain module, a missing service class or due to the service class not having `Siena.Server.API.Service` as base class.

`XMLRPC_SERVICE_METHOD_NOT_FOUND = 12`

The requested service method (Python class method) was not found.

This can be due to a misspelling of the service method, a missing mention of the method name in the `.public_methods` tuple of the service class or a missing method implementation.

`XMLRPC_BAD_REQUEST = 20`

The server could not decode the incoming XML-RPC request.

This is usually caused by a non-standard protocol extension being used, e.g. if the client uses some XML-RPC library which uses non-standard data types in the method call.

`XMLRPC_BAD_RESPONSE = 21`

The server could not encode the method return value into valid XML-RPC.

Since XML-RPC only supports a very limited range of data types, this error can be caused by using an unsupported data types in the response, e.g. a complex number or an unregistered extension type.

`XMLRPC_ACCESS_DENIED = 30`

Access to the requested service was not granted by the system.

This is either caused by a missing login of the client application, a missing license for the requested service or simply missing permissions for the requested action.

A service implementation can also cause this response by raising a `Siena.Server.API.Service.AccessForbiddenError`. See section 0 for details.

`XMLRPC_REQUEST_CANCELLED = 31`

Processing of the requested service was cancelled due to a problem in the service.

This is usually done under service control, so the service implementation defines the cause of this error. The exception needed to be raised to produce this response is

`Siena.Server.API.Service.CancelError`. See section 0 for details.

`XMLRPC_UNEXPECTED_EXCEPTION = 40`

The service request caused a Python exception.

These exceptions can be due to problems in the service method, an attached database or programming errors. Other possibilities are exceptions which called APIs raise to inform the caller of e.g. missing data sets or value range errors.

---

## 8.4 Standard “server.\*” Domain Services and Methods

The Siena Enterprise Server provides a set of pre-defined services in the `server.*` external name domain which can be queried for various server side tasks like service and method introspection, automated service reload, etc.

The following services and methods are available:

```
server.system.getServices()
```

Returns a list of available external service names as strings.

```
server.system.getMethods(servicename)
```

Returns a list of public methods defined for the service `servicename`. `servicename` has to be an external service name. A `KeyError` is raised in case the service is unknown. The `KeyError` is then handled using the conventions defined for the interface protocol.

```
server.system.reloadServices()
```

Initiates a service reload on the server.

Note that this can take a few seconds to propagate to all running service providing processes on the server.

Returns 1/0 depending on success of broadcasting the request.

This method is only available if the server is running in service debug mode, i.e. the `Config/system.ini` file must have the entry:

```
[Services]
debug = 1
```

---

## 8.5 Examples

This section demonstrates a few examples of Siena-based services to get started in developing your own services.

---

### 8.5.1 Simple implementations

The next few sections provide various examples of how to implement services. We introduce these examples by looking at the pre-installed service domain module [DemoServices.py](#) which you can find in the [Lib/UserServices/](#) directory on the server.

The module start with importing the main Siena Service API interface: the `Service` module:

```
from Siena.Server.API import Service
```

#### Defining a Standard Service Class

Every service class starts with sub-classing the `Service` base class from the `Siena.Server.API.Service` module.

```
class Demo(Service.Service):
    """ Demo service implementation.

    Note: You should *not* define an __init__() method since
    this is used by the Service base class. Instead, put the
    service startup code into the .startup() method which is
    called by __init__() right after the object has been
    setup.

    """
```

Next, we define the `.public_methods` attribute which has to be a tuple holding the names of all published methods.

Note that in this demonstration we extend this tuple as we move ahead. This is perfectly valid in Python, but should not be used for production quality code since it can be confusing not to have the public methods all listed in one place.

```
# Tuple of published method names
```

```
public_methods = ()
```

### Get the current date and time

```
def getCurrentTime(self):  
  
    """ Return the current date and time as string.  
    """  
    import time  
    return time.ctime(time.time())  
  
public_methods = public_methods + ('getCurrentTime',)
```

### Upper case a string

```
def upperCase(self, string):  
  
    """ Return the input string with all upper case characters.  
    """  
    return string.upper()  
  
public_methods = public_methods + ('upperCase',)
```

### Scientific calculations are possible

This example shows how Python can handle different input formats. This service method can work on strings as well as integers and floats.

```
def calcSine(self, value):  
  
    """ Return the sine of the input number as float.  
  
    This method also accepts strings as input. It  
    auto-converts them to floats before calculating the  
    sine.  
  
    """  
    import math  
    return math.sin(float(value))  
  
public_methods = public_methods + ('calcSine',)
```

### Simply math

```
def sumTwo(self, a, b):  
  
    """ Return the sum of the two input numbers a and b.  
    """  
    return a + b  
  
public_methods = public_methods + ('sumTwo',)
```

## Standard I/O is possible too...

...though not recommended as this can easily introduce security problems, if not carefully implemented.

```
base_directory = '/tmp/test/'

def readfile(self, filename):

    """ Read a file from the .base_directory directory and
        return the contents as binary data.

        If the filename does not exist, an exception is raised.

    """
    data = open(self.base_directory + filename, 'rb').read()
    return binary(data)

public_methods = public_methods + ('readfile',)
```

---

### 8.5.2 More complex implementations

The next few examples demonstrate handling of more sophisticated tasks. As you will see, these are really not much more elaborate than the simple ones above.

#### Reach out to the Internet

This little service method turns Siena into a web client (rather than a server).

```
def fetchURL(self, url):

    """ Read the given URL and return the contents found there
        as binary.

    """
    import urllib
    data = urllib.urlopen(url).read()
    return binary(data)

public_methods = public_methods + ('fetchURL',)
```

#### Read data from an attached database

In this example we read data from the attached system database and read a row from the table which stores the visitor data.

Since services are defined in terms of classes, we can easily put configuration data into the class definition:

```

# Name of the database connection to use in .readDatabase()
connection_name = 'system'

# Name of the table to query
table_name = 'siena_visitors'

```

This method uses Python an optional parameter: the row index.

```

def readDatabase(self, rowindex=1):

    """ Read a row from a connected database and return it as
    sequence.

    rowindex can be given to scroll forward in the result
    set.

    """
    connection = self.connection(self.connection_name)
    cursor = connection.cursor()
    cursor.execute('select * from %s' % self.table_name)
    for i in range(rowindex):
        row = cursor.fetchone()
    return row

public_methods = public_methods + ('readDatabase',)

```

### Implement a Domain Service

This domain service allows introspection of the available services in this domain module.

```

class DemoServices(Service.DomainService):

    """ Domain service implementation.

    There may only be one such class per module.

    """
    # Tuple of published method names
    public_methods = ('services',)

    ### Published Service API

    def services(self):
        return ('service',)

```

---

## 8.6 Debugging User Services

Debugging services written by the service author is aided in several ways. This section describes a few approaches which should simplify the task.

---

## 8.6.1 Server Side Debugging

Siena supports the developer in various ways. The top priority is making as much information available to the developer as possible.

### System Log File

The most important resource for finding information related to errors, bugs or other interesting notices is the system's central log file: [System/system.log](#) in the Siena installation directory.

To view the file, use your preferred text editor or a text viewer like `more` or `less`. You should not change the file, since it is constantly updated by the server to reflect the latest changes.

In case the file does not appear to be updated anymore, it is likely that the original file opened by the server was changed and that you are looking at an older version. The only way to fix this rare situation is by restarting the Siena Server.

To view the file, open a shell window on the server and then execute e.g.

```
su - siena
less -f System/system.log
```

This will constantly update the display of the log file as it is written by the Siena server.

### Printing to the Log File

You may sometimes want to save information while processing a service for later review and to aid in debugging a service.

There are several ways to do this, the easiest being to simply `print` the information using the standard Python `print` statement. The information will then be redirected to the log file and appear with the log level `0000`.

#### Example:

```
def method(self, arg):
    print 'method was called with %s' % repr(arg)
    return 'hello world!'
```

## Service Debug Mode

To run Siena in service debug mode, you have to edit the server's configuration file [Config/system.ini](#) and make sure that the following entry is set in the file:

```
[Services]
debug = 1
```

Restarting the server with this flag set will enable service debug mode.

In service debug mode, the following changes become active:

- the service method `server.system.reloadServices()` is available for calling from any client,
- fault messages generated by the interfaces will include the complete Python traceback of the error in their fault string instead of just a short notice.

---

### 8.6.2 Client Side Debugging

Siena does not come with its own client side development environment. There are still a few ways to retrieve debugging information from the server:

#### Break Points

One way is by inserting using exceptions to insert break points in the service implementation, e.g.

Example:

```
def method(self, arg):
    raise Service.CancelError, \
        'method was called with %s' % repr(arg)
    # this line is never reached
    return 'hello world!'
```

#### Logging to the session object

Since the Siena Enterprise Server supports session management it is also possible to log the debug information in the session object and then have other service methods retrieve this information later on.

### **Logging to an attached database**

A similar approach is possible by storing debug information to an attached database and then using client side database tools to extract the information.

### **Client-side development environment**

Service implementation can be developed on the local client using the Siena Personal Server. See chapter 3 for details.

---

## 9. Frequently Asked Questions (FAQ)

This section explains some commonly asked questions about the system in a question/answer style.

---

### 9.1 Services

This section includes questions related to writing services and installing them.

---

#### 9.1.1 None of the installed services work anymore and the system.log contains a note about UserServices. How can this be fixed ?

You have probably accidentally deleted the file `__init__.py` which is needed by Python to recognise a file system directory as Python package. The solution is to create an empty file `__init__.py` in the `Lib/UserServices/` directory and restart the server.

---

#### 9.1.2 My service xyz doesn't work. What can I do ?

If you have followed the service module installation instructions closely, then the Siena server will be able to find the module. However, if the server cannot import the module because of a `SyntaxError` or some other startup time error in the module, it will only print a note to the `System/system.log` file and ignore the module.

Please check the `System/system.log` file for such a message and correct the problem mentioned there in the module.

---

#### 9.1.3 How do I create a new service domain ?

As `siena` user, go to the `Lib/UserServices/` directory, create a directory having the domain name (without any spaces or other special characters) and add an empty `__init__.py` file to the directory. Next, add service

modules to this new directory and reload the server. The services in your service modules will then become available under the new domain.

**Note:**

The domain directory must be readable and writeable by the *siena* user account. If it is not writeable, then Siena cannot create a compiled copy of the service modules (.pyo files) and performance will go down.

---

## 10. Obtaining more information

More information about the server and available components, support and extensions is available at <http://www.siena-tech.com/>.

If you should need more information about the **Python Programming Language** and its resources, please visit the web-sites <http://www.python.org/> and <http://starship.python.net/>.

**Announcements** regarding the system and its development will be posted to the following mailing list (among others):  
announcements@siena-tech.com

To subscribe, send a message having *Subscribe* as body text to the following address: [announcements-request@siena-tech.com](mailto:announcements-request@siena-tech.com).

---

# Appendix A

---

## Connection objects

### *Methods:*

`.cursor()`

Returns a cursor object usable on the connection. Cursor objects have several methods that depend on the database interface used, e.g. for the ODBC interface the methods are defined by the `mx.ODBC` extensions package.

`.commit()`

Commits all changes done on the connection. Note that the Siena automatically commits the changes after every successfully processed request and send rollbacks in case of an unsuccessful request.

`.rollback()`

Rolls back all changes done on the connection. Also see note for `.commit()` above.

---

## Cursor objects

These objects represent a database cursor, which is used to manage the context of a fetch operation. For database interfaces that comply with the Python Database API, e.g. `mx.ODBC` (the ODBC interface used by Siena), they provide (at least) the following methods and attributes:

### *Attributes:*

`.description`

This read-only attribute is a sequence of 7-item sequences. Each of these sequences contains information describing one result column: `(name, type_code, display_size, internal_size, precision, scale, null_ok)`. This attribute will be `None` for operations that do not return rows or if the cursor has not had an operation invoked via the `execute()` method yet.

The `type_code` is equal to one of the type objects specified by the interface module.

`.rowcount`

This read-only attribute specifies the number of rows that the last `execute()` produced (for DQL statements like `SELECT`) or affected (for DML statements like `update` or `INSERT`). The attribute is 0 in case no `execute()` has been performed on the cursor.

`.arraysize`

The number of rows to fetch in a single call to `.fetchmany()` when the size parameter is not given.

### **Methods:**

`.close()`

Close the cursor now (rather than whenever `__del__` is called). The cursor will be unusable from this point forward; an exception will be raised if any operation is attempted with the cursor.

`.execute(operation[,params])`

Prepare and execute a database operation (query or command). Parameters may be provided (as tuple) and will be bound to variables in the operation. Variables are specified in a database-specific notation (some DBs use `?, ?, ?` to indicate parameters, others `:1, :2, :3`; mx.ODBC uses `?, ?, ?` as parameter markers) that is based on the index in the parameter tuple (position-based rather than name-based).

The parameters may also be specified as a list of tuples to e.g. insert multiple rows in a single operation.

A reference to the operation will be retained by the cursor. If the same operation object is passed in again, then the cursor can optimize its behavior. This is most effective for algorithms where the same operation is used, but different parameters are bound to it (many times).

`.fetchone()`

Fetch the next row of a query result set, returning a single tuple, or `None` when no more data is available.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

`.fetchmany([size])`

Fetch the next set of rows of a query result, returning as a list of tuples. An empty list is returned when no more rows are available. The number of rows to fetch is specified by the parameter. If it is `None`, then the cursor's `arraysize` attribute determines the number of rows to be fetched.

Note there are performance considerations involved with the size parameter. For optimal performance, it is usually best to use the `arraysize` attribute. If the size parameter is used, then it is best for it to retain the same value from one `fetchmany()` call to the next.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.

`.fetchall()`

Fetch all (remaining) rows of a query result, returning them as a list of tuples. Note that the cursor's `arraysize` attribute can affect the performance of this operation.

An exception is raised if the previous call to `execute()` did not produce any result set or no call was issued yet.