TEAMCENTER

Teamcenter Services

Teamcenter 2412

SIEMENS

About Siemens Digital Industries Software

Siemens Digital Industries Software is a global leader in the growing field of product lifecycle management (PLM), manufacturing operations management (MOM), and electronic design automation (EDA) software, hardware, and services. Siemens works with more than 100,000 customers, leading the digitalization of their planning and manufacturing processes. At Siemens Digital Industries Software, we blur the boundaries between industry domains by integrating the virtual and physical, hardware and software, design and manufacturing worlds. With the rapid pace of innovation, digitalization is no longer tomorrow’s idea. We take what the future promises tomorrow and make it real for our customers today. Where today meets tomorrow. Our culture encourages creativity, welcomes fresh thinking and focuses on growth, so our people, our business, and our customers can achieve their full potential.

Support Center: **support.sw.siemens.com**

Send Feedback on Documentation: **support.sw.siemens.com/doc_feedback_form**

Contents

1. Getting started with Teamcenter Services

Before you begin

Prerequisites	The following are required to use Teamcenter Services: • Access to a Teamcenter server • An application development environment used for Java, C++, or .NET software development • Java JDK for Java development • Visual Studio for C++ development For information about system hardware and software requirements for Teamcenter, see the Siemens Digital Industries Software Certification Database: **http://support.industrysoftware.automation.siemens.com/certification/teamcenter.shtml**
Verify access to services	Ensure your Teamcenter environment is **set up to access Teamcenter Services**.
Test services access	**Run sample applications** to test your access to Teamcenter Services. You can use the sample client applications as a guideline to create your own clients.
Configure custom client	You must **configure your client to use Teamcenter Services'** client libraries. No configuration of the Teamcenter server is required.

First steps with Teamcenter Services

You can use Teamcenter Services to create loosely coupled integrations with Teamcenter, regardless of the programming language standards and operating environments prevalent in your organization. Services can be used by Teamcenter client applications, other Siemens Digital Industries Software applications, and customers' in-house applications.

Teamcenter Services use service-oriented architecture (SOA). SOA allows for contract-based, coarse-grained interfaces to expose Teamcenter functionality in a standard way. Teamcenter Services provide service interfaces that are both language and technology independent. All applications can use the same interfaces to backend systems, resulting in lower system and maintenance costs.

You can use Teamcenter Services to connect your company's applications to Teamcenter operations, such as create, checkin, checkout, and so on.

You can see all the available Teamcenter operations that you can connect to in the *Services Reference*.

> Note:
>
> The *Services Reference* is available on Support Center.

To get started using sample applications so you can learn how to connect your company's applications to Teamcenter operations, perform the following steps:

1. Ensure that your Teamcenter environment meets the necessary prerequisites.
2. Set up your Teamcenter environment so that you run Teamcenter Services.
3. Install the **HelloTeamcenter** sample application.
4. Examine the **HelloTeamcenter** sample application to see how Teamcenter Services work.

Enable Teamcenter services

Process for accessing Teamcenter Services

Teamcenter Services are an integral architectural component of Teamcenter, regardless of the deployment configuration (two-tier or four-tier) selected. The language-specific client libraries you must import into your development environment are available on the Teamcenter installation software distribution image (for example, the installation DVD) in the **soa_client.zip** file. Libraries are included for Java, C++, and C# (.NET), as well as WSDL interfaces. All Teamcenter Services WSDLs are compliant with the WS-I Basic Profile.

Perform different steps depending on whether you are accessing Teamcenter Services in a four-tier or two-tier environment:

- Four-tier environment

  If you want to use WSDL based clients, you must also install the **Teamcenter Services WSDL/SOAP Support** module.

  1. Ensure that your Teamcenter server manager is running and that you have network access to it.
  2. Verify that you can access Teamcenter Services by running the **HelloTeamcenter** sample application using the host argument set to either:

     - The address of the Teamcenter Web Tier server.

http://<web host>:7001/tc

- The TCCS environment name.

  tccs://Teamcenter-4Tier

- Two-tier environment

  1. Ensure that a Teamcenter 2-tier configuration is installed on the local machine. TCCS will start the tcserver process as needed.

  2. Verify that you can access Teamcenter Services by running the **HelloTeamcenter** sample application with argument set to the TCCS environment name.

     tccs://Teamcenter-2Tier

Run the HelloTeamcenter sample application

To become familiar with Teamcenter Services, you should examine the **HelloTeamcenter** sample application.

Sample client applications are provided in the **soa_client.zip** file on the Teamcenter software distribution image. The **soa_client.zip** file contains the **cpp**, **java**, and **net** folders, each of which contains a **samples** folder with sample client applications for each language binding. To run the sample client applications, follow the instructions in the **ReadMe.txt** files provided with each sample application.

he following procedure is only an example of how to run a sample client application. For complete instructions, see the **ReadMe.txt** file.

To run the **HelloTeamcenter** sample application from the **cpp**, **java**, or **net** folders, perform the following general steps:

1. Verify that your Teamcenter server is running.

2. Install Eclipse if you want to run a Java sample application, or install Microsoft Visual Studio if you want to run a C++ or C# (.NET) sample application.

   > Note:
   >
   > For information about system hardware and software requirements for Teamcenter, see the Hardware and Software Certifications knowledge base article on Support Center.

3. Set up your Eclipse or Visual Studio environment.

   - Eclipse

Add a classpath variable for the root location of the **soa_client** folder:

a. Choose **Window→Preferences**.

b. In the **Preferences** dialog box, choose **Java→Build Path→Classpath Variables**.

c. In the **Classpath Variables** dialog box, click **New** and add the **TEAMCENTER_SERVICES_HOME** variable with its path set to the **soa_client** folder.

- Visual Studio

  If you are running a C++ sample application, add the location of the libraries to the **PATH** environment variable:

  ```
  SET PATH=install-location\soa_client\cpp\libs\wnti32;%PATH%
  ```

4. Import the sample application project.

   - Eclipse

     a. Choose **File→Import**.

     b. In the **Import** dialog box, choose **General→Existing Projects into Workspace**.

     c. Click **Next**.

     d. In the **Select root directory** box, click **Browse**, and then browse to **soa_client/java/samples/HelloTeamcenter**.

     e. Click **Finish**.

   - Visual Studio

     a. Choose **File→Open→Project/Solution**.

     b. For the C++ sample application browse to:

     ```
     install-location/soa_clients/cpp/samples/HelloTeamcenter/HelloTeamcenter.vcproj
     ```

     For the NET sample application, browse to:

     ```
     install-location/soa_clients/net/samples/HelloTeamcenter/HelloTeamcenter.csproj
     ```

5. Ensure the project is compiled.

Right-click the **HelloTeamcenter** project and choose **Build Project** (Eclipse) or **Build** (Visual Studio).

6. Execute the client application.

   a. Choose **Run→Debug Configurations** (Eclipse) or **Debug→Start Without Debugging** (Visual Studio) and select the **HelloTeamcenter** application. For example, if you are running the Java version, choose **Java Application→HelloTeamcenter**.

      The client application runs in the **Console** view (Eclipse) or a new window (Visual Studio).

      > Tip:
      >
      > By default, the sample applications are configured to run in a four-tier environment. If your environment is a two-tier environment, you must change the setting on the **Connection** object in the sample application to switch from four-tier to two-tier.

      For example, if you are running the Java example in Eclipse, choose **Java Application→HelloTeamcenter**, click the **Arguments** tab, and in the **VM arguments** box, change:

      ```
      -Dhost=http://host-name:7001/tc
      ```

      to:

      ```
      -Dhost=tccs//environment-name
      ```

      Replace *environment-name* with the TCCS environment name.

   b. Log on to Teamcenter to see data returned from the server.

Now that you have run the **HelloTeamcenter** sample application, a **detailed explanation** of the sample application may be helpful.

HelloTeamcenter sample application walk-through

Following is a walk-through of the **HelloTeamcenter** sample Java application. After reading this topic, you should have a basic understanding of the structure of client applications that use Teamcenter Services, and you should be able to start writing your own client applications.

This sample application demonstrates some of the basic features of the Teamcenter Services framework and a few of the services. The application:

1. Creates an instance of the **Connection** object with implementations of the **ExceptionHandler**, **PartialErrorListener**, **ChangeListener**, and **DeleteListeners** interfaces.

2. Establishes a session with the Teamcenter server.

3. Displays the contents of the **Home** folder.

4. Performs a simple query of the database.

5. Creates, revises, and deletes an item.

Although the sample application performs only a few tasks, it shows the basic Teamcenter Services approach that all client applications follow. Walk through the code to find out how it works:

1. Look at the following code from the **Hello.java** class in the **HelloTeamcenter** sample Java application. Note how it performs each step in the task:

```
//==================================================
//
//  Copyright 2012 Siemens Product Lifecycle Management Software Inc. All
Rights
    Reserved.
//
//==================================================

package com.teamcenter.hello;

import com.teamcenter.clientx.Session;
import com.teamcenter.soa.client.model.strong.User;



/**
 * This sample client application demonstrates some of the basic features
of the
 * Teamcenter Services framework and a few of the services.
 *
 * An instance of the Connection object is created with implementations of
the
 * ExceptionHandler, PartialErrorListener, ChangeListener, and
DeleteListeners
 * intefaces. This client application performs the following functions:
 * 1. Establishes a session with the Teamcenter server
 * 2. Display the contents of the Home Folder
 * 3. Performs a simple query of the database
 * 4. Create, revise, and delete an Item
 *
 */
public class Hello
{
```

```
    /**
     * @param args   -help or -h will print out a Usage statement
     */
    public static void main(String[] args)
    {
        if (args.length > 0)
        {
            if (args[0].equals("-help") || args[0].equals("-h"))
            {
                System.out.println("usage: java [-Dhost=http://
server:port/tc]
                    com.teamcenter.hello.Hello"); System.exit(0);
            }
        }

        // Get optional host information
        String serverHost = "http://localhost:7001/tc";
        String host = System.getProperty("host");
        if (host != null && host.length() > 0)
        {
            serverHost = host;
        }



        Session   session = new Session(serverHost);
        HomeFolder   home = new HomeFolder();
        Query       query = new Query();
        DataManagement dm = new DataManagement();



        // Establish a session with the Teamcenter Server
        User user = session.login();

        // Using the User object returned from the login service request
        // display the contents of the Home Folder
        home.listHomeFolder(user);

        // Perform a simple query of the database
        query.queryItems();

        // Perform some basic data management functions
        dm.createReviseAndDelete();

        // Terminate the session with the Teamcenter server
        session.logout();

    }
```

```
}
```

2. To accomplish each step in the task (display folder contents, perform a query, and create items) the **Hello.java** class calls methods located in the other Java classes in the sample application: **HomeFolder.java**, **Query.java**, and **DataManagement.java**.

   Follow the code to see how services are called for each task. For example, to see how items are created using a service operation, look at the following code from the **DataManagement.java** class. Start with the **createReviseAndDelete** method, which in turn calls the **createItems** service operation.

```
public void createReviseAndDelete()
    {
        try
        {
            int numberOfItems = 3;

            // Reserve Item IDs and Create Items with those IDs
            ItemIdsAndInitialRevisionIds[] itemIds = generateItemIds
                (numberOfItems, "Item");
            CreateItemsOutput[] newItems = createItems(itemIds, "Item");

            // Copy the Item and ItemRevision to separate arrays for further
            // processing
            Item[] items = new Item[newItems.length];
            ItemRevision[] itemRevs = new ItemRevision[newItems.length];
            for (int i = 0; i < items.length; i++)
            {
                items[i] = newItems[i].item;
                itemRevs[i] = newItems[i].itemRev;
            }

            // Reserve revision IDs and revise the Items
            Map<BigInteger,RevisionIds> allRevIds =
generateRevisionIds(items);
            reviseItems(allRevIds, itemRevs);

            // Delete all objects created
            deleteItems(items);
        }
        catch (ServiceException e)
        {
            System.out.println(e.getMessage());
        }

    }
.
.
```

```
.
    /**
     * Create Items
     *
     * @param itemIds        Array of Item and Revision IDs
     * @param itemType       Type of item to create
     *
     * @return Set of Items and ItemRevisions
     *
     * @throws ServiceException  If any partial errors are returned
     */
    @SuppressWarnings("unchecked")
    public CreateItemsOutput[] createItems(ItemIdsAndInitialRevisionIds[]
        itemIds, String itemType)
            throws ServiceException
    {
        // Get the service stub
        DataManagementService dmService = DataManagementService.getService
            (Session.getConnection());
        // Populate form type
        GetItemCreationRelatedInfoResponse relatedResponse = dmService.
            getItemCreationRelatedInfo(itemType, null);
        String[] formTypes = new String[0];
        if ( relatedResponse.serviceData.sizeOfPartialErrors() > 0)
            throw new ServiceException( "DataManagementService.
                getItemCretionRelatedInfo returned a partial error.");

        formTypes = new String[relatedResponse.formAttrs.length];
        for ( int i = 0; i < relatedResponse.formAttrs.length; i++ )
        {
            FormAttributesInfo attrInfo = relatedResponse.formAttrs[i];
            formTypes[i] = attrInfo.formType;
        }

        ItemProperties[] itemProps = new ItemProperties[itemIds.length];
        for (int i = 0; i < itemIds.length; i++)
        {
            // Create form in cache for form property population
            ModelObject[] forms = createForms(itemIds[i].newItemId,
formTypes[0],
                                              itemIds[i].newRevId,
formTypes[1],
                                              null, false);
            ItemProperties itemProperty = new ItemProperties();

            itemProperty.clientId = "AppX-Test";
            itemProperty.itemId = itemIds[i].newItemId;
            itemProperty.revId = itemIds[i].newRevId;
            itemProperty.name = "AppX-Test";
```

```
            itemProperty.type = itemType;
            itemProperty.description = "Test Item for the SOA AppX sample
                application.";
            itemProperty.uom = "";

            // Retrieve one of form attribute value from Item master form.
            ServiceData serviceData = dmService.getProperties(forms,
                new String[]{"project_id"});
            if ( serviceData.sizeOfPartialErrors() > 0)
                throw new
ServiceException( "DataManagementService.getProperties
                returned a partial error.");
            Property property = null;
            try
            {
                property= forms[0].getPropertyObject("project_id");
            }
            catch ( NotLoadedException ex){}


            // Only if value is null, we set new value
            if ( property == null || property.getStringValue() ==
                null || property.getStringValue().length() == 0)
            {
                itemProperty.extendedAttributes = new ExtendedAttributes[1];
                ExtendedAttributes theExtendedAttr = new
ExtendedAttributes();
                theExtendedAttr.attributes = new HashMap<String,String>();
                theExtendedAttr.objectType = formTypes[0];
                theExtendedAttr.attributes.put("project_id", "project_id");
                itemProperty.extendedAttributes[0] = theExtendedAttr;
            }
            itemProps[i] = itemProperty;
        }


        // ****************************
        // Execute the service operation
        // ****************************
        CreateItemsResponse response = dmService.createItems(itemProps,
null, "");
        // before control is returned the ChangedHandler will be called with
        // newly created Item and ItemRevisions



        // The AppXPartialErrorListener is logging the partial errors
returned
        // In this simple example if any partial errors occur we will throw
```

a

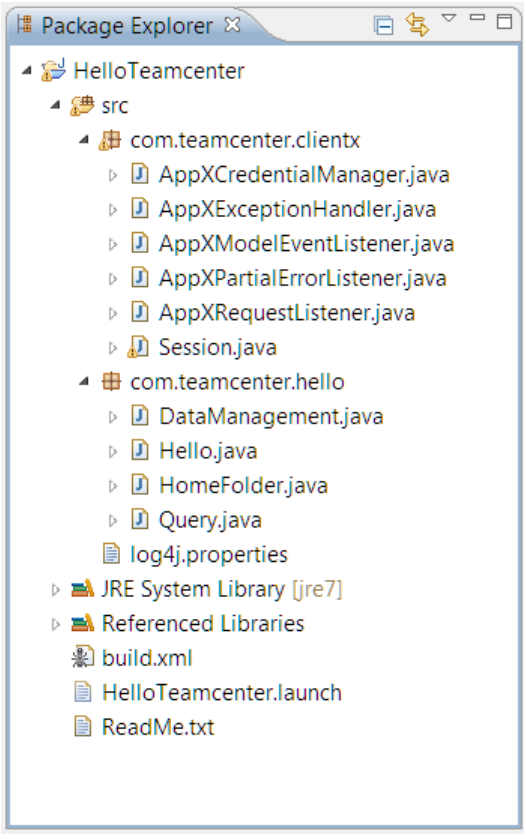
```
        // ServiceException
        if (response.serviceData.sizeOfPartialErrors() > 0)
            throw new ServiceException( "DataManagementService.createItems
            returned a partial error.");

        return response.output;
    }
```

3. After examining the code, perform the following steps to run the **HelloTeamcenter** sample application Java project:

    a. Import the **HelloTeamcenter** sample application Java project into Eclipse.

    b. Examine the files.

    When you import the **HelloTeamcenter** sample application Java project into Eclipse, it appears as follows.



HelloTeamcenter sample application Java project in Eclipse

The project is made up of the following components:

- **com.teamcenter.clientx** package

  Contains the implementation classes corresponding to the following Teamcenter Services framework interfaces.

Implementation class	Interface
AppXCredentialManager	**CredentialManager**
AppXExceptionHandler	**ExceptionHandler**
AppXModelEventListener	**ChangeListener**
AppXPartialErrorListener	**PartialErrorListener**
AppXRequestListenerHandler	**RequestListener**

  To fulfill the *contract-based interfaces* of the Teamcenter Services framework, these classes must provide all the required methods defined for the appropriate interface.

  Also in this package, the **Session** class demonstrates how to establish a session using Teamcenter Services by using the **SessionService login()** and **logout()** methods. A valid session must first be established before any Teamcenter Services operations are called.

- **com.teamcenter.hello** package

  Contains the classes that demonstrate making various Teamcenter Services calls (once the session is established) in the following areas:

  - **DataManagement** creates, revises, and deletes a set of items.
  - **Hello** contains the main method, or entry point to the application, and is the best place to begin browsing the code.
  - **HomeFolder** lists the contents of the user's home folder.
  - **Query** performs a simple query.

  All the classes in this package perform the required basic four steps to interact with Teamcenter Services.

  A. Construct the desired service stub.

  B. Gather the data for the operation's input arguments.

  C. Call the service operation

  D. Process the results.

- **build.xml** file (build instructions)

  Contains information describing the internal build targets required to build the Eclipse project.

- **HelloTeamcenter.launch** file (launch configuration)

  Contains information telling Eclipse how to launch this client application, including its default argument settings. A single argument (a string) defines the HTTP or TCCS address to use for the connection to the Teamcenter server.

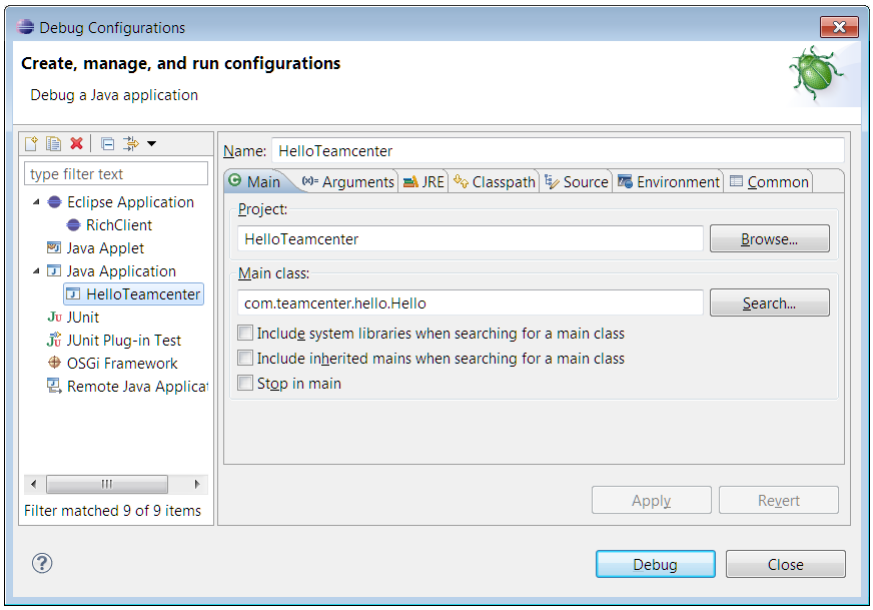
- **readme.txt** file

  Describes the purpose and contents of the project and its files.

c. Execute the client application.

A. Ensure your Teamcenter environment is enabled to run Teamcenter Services.

B. In Eclipse, choose **Run→Debug Configurations**. Choose **Java Application→HelloTeamcenter**, and then click **Debug** at the bottom of the dialog box.



Running the HelloTeamcenter sample application Java application

The client application runs in the **Console** view.

C. In the **Console** view, you are prompted to enter a Teamcenter user name and password.

The application displays the data returned from the server, including the contents of the **Home** folder. Note the services that are called, for example, **Session.login**, **DataManagement.getProperties**, and so on:

```
Please enter user credentials (return to quit):
User Name: infouser
Password:  infouser
SVI6W181.12951.01.Anonymous.00001: Core-2008-06-Session.login
SVI6W181.12951.01.Anonymous.00001.01.infouser.00002: Core-2007-01-Session.getTCSessionInfo
SVI6W181.12951.01.Anonymous.00001.01.infouser.00002.01.infouser.00003: Core-2011-06-
Session.getTypeDescriptions
SVI6W181.12951.01.Anonymous.00001.01.infouser.00004: Core-2011-06-Session.getTypeDescriptions
SVI6W181.12951.01.infouser.00005: Core-2006-03-DataManagement.getProperties
SVI6W181.12951.01.infouser.00005.01.infouser.00006: Core-2011-06-Session.getTypeDescriptions

Modified Objects handled in com.teamcenter.clientx.AppXUpdateObjectListener.modelObjectChange
The following objects have been updated in the client data model:
    gcRBx4VkI1ITGA Fnd0ClientSessionInfo

Home Folder:
SVI6W181.12951.01.infouser.00007: Core-2006-03-DataManagement.getProperties
Name        Owner        Last Modified
====        =====        =============
ClientCache    info    7/28/2012 1:04 PM
MetaDataStamp Templates    infouser    7/28/2012 12:59 PM
RequirementsManagement Templates    infouser    7/28/2012 12:59 PM
MS Office Templates    infouser    7/28/2012 12:57 PM
CAM Machining Knowledge    infouser    7/28/2012 12:56 PM
CAM Express Tutorials    infouser    7/28/2012 12:56 PM
CAM Setup Templates    infouser    7/28/2012 12:56 PM
Unigraphics UDF parts    infouser    7/28/2012 12:56 PM
Unigraphics seed parts    infouser    7/28/2012 12:56 PM
Mailbox    infouser    7/28/2012 12:45 PM
Newstuff    infouser    7/28/2012 1:01 PM
My Saved Searches    infouser    8/2/2012 12:21 PM
SVI6W181.12951.01.infouser.00008: Query-2006-03-SavedQuery.getSavedQueries
SVI6W181.12951.01.infouser.00008.01.infouser.00009: Core-2011-06-Session.getTypeDescriptions
SVI6W181.12951.01.infouser.00010: Query-2008-06-SavedQuery.executeSavedQueries

Found Items:
SVI6W181.12951.01.infouser.00011: Core-2007-09-DataManagement.loadObjects
SVI6W181.12951.01.infouser.00011.01.infouser.00012: Core-2011-06-Session.getTypeDescriptions
SVI6W181.12951.01.infouser.00013: Core-2006-03-DataManagement.getProperties
Name        Owner        Last Modified
====        =====        =============
cap_screw_cs_metric    infouser    7/28/2012 12:56 PM
cam_english_template    infouser    7/28/2012 12:56 PM
Turning_Exp_inch    infouser    7/28/2012 12:56 PM
Turning_Express_target_metric    infouser    7/28/2012 12:56 PM
Machinery_Express_target_inch    infouser    7/28/2012 12:56 PM
Machinery_Express_target_metric    infouser    7/28/2012 12:56 PM
shops_die_sequences_metric    infouser    7/28/2012 12:56 PM
MultiAxis_Exp_metric    infouser    7/28/2012 12:56 PM
mill_multi_blade_metric    infouser    7/28/2012 12:56 PM
legacy_lathe_inch    infouser    7/28/2012 12:56 PM
SVI6W181.12951.01.infouser.00014: Core-2007-09-DataManagement.loadObjects
SVI6W181.12951.01.infouser.00014.01.infouser.00015: Core-2011-06-Session.getTypeDescriptions
SVI6W181.12951.01.infouser.00016: Core-2006-03-DataManagement.getProperties
Name        Owner        Last Modified
```

```
====        =====        =============
Turning_Express_inch    infouser    7/28/2012 12:56 PM
Turning_Express_metric    infouser    7/28/2012 12:56 PM
RSWP_3T__84d8c460    infouser    7/28/2012 12:57 PM
WELD_1__84d8c460    infouser    7/28/2012 12:57 PM
WELD_38__84d8c460    infouser    7/28/2012 12:57 PM
WELD_41__84d8c460    infouser    7/28/2012 12:57 PM
WELD_46__84d8c460    infouser    7/28/2012 12:57 PM
WELD_48__84d8c460    infouser    7/28/2012 12:57 PM
000001-REQ_default_spec_template    infouser    7/28/2012 12:59 PM
000002-REQ_default_spec_template_for_view    infouser    7/28/2012 12:59 PM
SVI6W181.12951.01.infouser.00017: Core-2007-09-DataManagement.loadObjects
SVI6W181.12951.01.infouser.00018: Core-2006-03-DataManagement.getProperties
Name        Owner        Last Modified
====        =====        =============
Metric    infouser    7/28/2012 12:56 PM
fit_hole_inch    infouser    7/28/2012 12:56 PM
standard_thread_metric    infouser    7/28/2012 12:56 PM
mill_multi_blade_inch    infouser    7/28/2012 12:56 PM
mill_planar_hsm_metric    infouser    7/28/2012 12:56 PM
SVI6W181.12951.01.infouser.00019: Core-2006-03-
DataManagement.generateItemIdsAndInitialRevisionIds
SVI6W181.12951.01.infouser.00020: Core-2007-01-DataManagement.getItemCreationRelatedInfo
SVI6W181.12951.01.infouser.00021: Core-2007-01-DataManagement.createOrUpdateForms
SVI6W181.12951.01.infouser.00021.01.infouser.00022: Core-2011-06-Session.getTypeDescriptions
SVI6W181.12951.01.infouser.00023: Core-2006-03-DataManagement.getProperties
SVI6W181.12951.01.infouser.00024: Core-2007-01-DataManagement.createOrUpdateForms
SVI6W181.12951.01.infouser.00025: Core-2006-03-DataManagement.getProperties
SVI6W181.12951.01.infouser.00026: Core-2007-01-DataManagement.createOrUpdateForms
SVI6W181.12951.01.infouser.00027: Core-2006-03-DataManagement.getProperties
SVI6W181.12951.01.infouser.00028: Core-2006-03-DataManagement.createItems

*****
Partial Errors caught in com.teamcenter.clientx.AppXPartialErrorListener.
Partial Error for client id AppX-Test
    Code: 38207    Severity: 3    Business rules require that you enter Description.
Partial Error for client id AppX-Test
    Code: 38207    Severity: 3    Business rules require that you enter Description.
Partial Error for client id AppX-Test
    Code: 38207    Severity: 3    Business rules require that you enter Description.
DataManagementService.createItems returned a partial error.
SVI6W181.12951.01.infouser.00029: Core-2006-03-Session.logout
```

4. (Optional) Run a different service from the **HelloTeamcenter** sample application Java project.

   There are many predefined services provided with Teamcenter that can be found listed in the *Services Reference*.

   > Note:
   >
   > The *Services Reference* is available on Support Center.

   Because the **HelloTeamcenter** sample application contains the basic framework of a client application, you can call another predefined Teamcenter service from the **HelloTeamcenter** sample application as a way to better understand how to create your own client application.

a. Clean the existing **Hello.java** class.

Below the **Session session ...** line, remove or comment out the following lines because you do not want to call the sample functions any more, but instead want to call different functions:

```
HomeFolder home = new HomeFolder();
Query query = new Query();
DataManagement dm = new DataManagement();
```

Below the **session.login();** line, remove or comment out these lines:

```
home.listHomeFolder(user);
query.queryItems();
dm.createReviseAndDelete();
```

b. Create new Java classes that call predefined services and reference them in the existing **Hello.java** class.

For additional examples about how predefined services are called, see other sample application in the **soa_client\java\samples** directory.

> Note:
>
> You can also create your own custom services and run them from the **HelloTeamcenter** sample application project to verify their operation.

Now that you have walked through the **HelloTeamcenter** sample application, you should be able to get started creating your own client applications using Teamcenter Services.

HelloTcOauth sample applications

Following is a walk-through of the **HelloTcOauth** sample applications. After reading this topic, you should have a basic understanding of the structure of how to use an OAuth2.0 access token to access Teamcenter Services.

Currently, the only fully supported OAuth provider is the SAM Auth service, which is part of the Security Services from Siemens Digital Industries Software.

Overview

The OAuth sample applications are provided in the *soa_client.zip* file in the *java\samples* directory. To build and run the sample client applications, follow the instructions in the *ReadMe.txt* files provided with each sample application.

Both applications demonstrate similar steps:

1. Initiate an OAuth login (based on the authorization code flow) with the configured OAuth provider.

2. Process the OAuth authorization code response and send a token request to the OAuth provider.

3. Process the OAuth token response and retrieve the OAuth access token.

4. Create an instance of the Teamcenter **Connection** object

5. Initiate a Teamcenter log in by passing a placeholder user ID and the OAuth access token rather than a real user ID and a password.

The source code for these applications can be inspected to more fully understand the steps involved.

The OAuth communication for these sample applications uses the third-party Java library *Nimbus OAuth 2.0 SDK with OpenID Connect extensions* created by **Connect2id**.

The sample applications require a Teamcenter 4-Tier server environment that is correctly configured to allow the OAuth-based login. This includes the following requirements:

1. The Teamcenter web tier is correctly configured for Security Services

2. The Teamcenter middle-tier servers are correctly configured for Security Services

3. Teamcenter Security Services Identity Service is deployed with:

    a. OAuth tokens enabled.

    b. A valid OAuth provider configured for OAuth token validation.

    c. A valid LDAP server configured for user ID mapping.

HelloTcOauth (Java)

Before building and running the HelloTcOauth sample, some basic configurations need to be set. These should be set in the *src\app.properties* file. Most of the properties can be left as default values unless there is a specific need to change them. The properties that will always need to be set are:

tc.url

This is the Teamcenter web tier URL. For example: http://host:7001/tc

oauth.client_id

This is the OAuth 2.0 client ID value received from the OAuth provider when the application is registered.

HelloTcOauth is a command line-based desktop application. When it runs, it will open a browser tab to initiate the OAuth login. It will also open a callback endpoint that listens for HTTP requests. The application follows the Proof Key for Code Exchange standard for authenticating the token request. Once

the OAuth login in the browser is completed successfully, the application will automatically connect to Teamcenter and initiate a log in using the OAuth Access Token. A successful Teamcenter log in will display similar output to the following, which lists the active connection information:

```
Connecting to Teamcenter...
Sending Login Request...
Login Response Received:
en_US
tcserver.exe3d90780d.syslog
tcserver00001@PoolA@18764@vc6s015
P14000.3.0.20230417.00
toltman
en_US
14.3:20230417.00
vc6s015
```

HelloTcOauthWeb (Jetty)

Before building and running the HelloTcOauthWeb sample, some basic configurations need to be set. These should be set in the *src\app.properties* file. Most of the properties can be left as default values unless there is a specific need to change them. The properties that will always need to be set are:

tc.url

Teamcenter web tier URL. For example: http://host:7001/tc

oauth.client_id

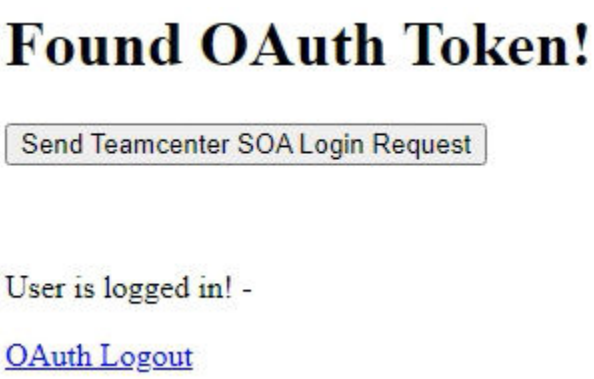
OAuth 2.0 Client ID value received from the OAuth provider when the application is registered.

oauth.client_secret

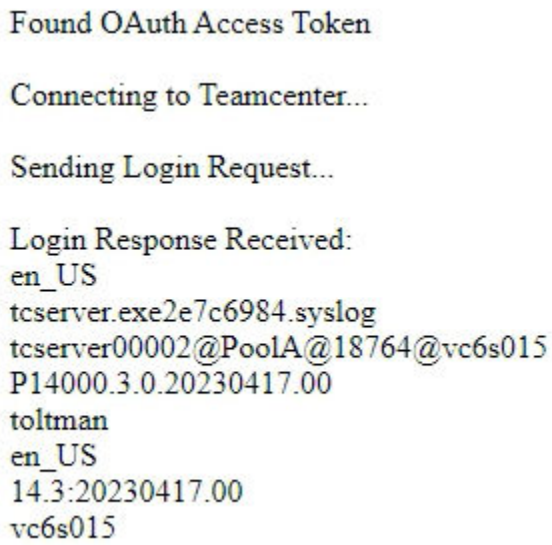
OAuth 2.0 Client Secret value received from the OAuth provider when the application is registered.

HelloTcOauthWeb is a simple Jetty web server application. When it runs, a browser will be needed to access the application. Once the server is started open a browser tab and navigate to the web server URL (**http://localhost:8855/** by default). If not already logged into the OAuth provider, the application will redirect the browser to complete the OAuth login.

Once the OAuth login in the browser has completed successfully, the browser displays the following to notify the user that it has successfully received an OAuth Access Token from the OAuth provider:



To initiate the Teamcenter connection and login, click the 'Send Teamcenter SOA Login Request' button shown on the web page. A successful Teamcenter login will return a page like the following, which lists the active connection information:



Creating services

Teamcenter Services are the best option available for clients to be able to access Teamcenter over a four-tier metadata architecture. Services are also the supported mechanism for integrating third-party software or writing a custom client. If the existing services provide all the functionality you need, you do not need to create a custom service.

However, if you want to access a custom business object operation, or run an ITK-style utility from your client, it is not possible without creating a service wrapper. The client calls the service, and the service either performs the requested action itself, or it passes on the call to another function.

You can create your own Teamcenter services and service operations using the Business Modeler IDE. The Business Modeler IDE is a tool for configuring the data model of your Teamcenter installation. To use

the Business Modeler IDE to create services, you must first set up a project. After you create services, you must test them by deploying them to a server.

When you use the Business Modeler IDE to create your own services and service operations, follow this process:

1. Add a service library.
2. Add a service.
3. Add service data types.
4. Add service operations.
5. Generate service artifacts.
6. Write implementations of the service operations.
7. Build server and client libraries.

After you create custom services, perform the following steps to call the custom services in your client application. (The following steps assume you are using Eclipse to create a Java client application. To verify these steps, you can use the **HelloTeamcenter** sample application to call your custom services.)

1. Add your custom services JAR files to your Eclipse project.

   In the Business Modeler IDE, the template project package is copied to a common location for deployment using Teamcenter Environment Manager (TEM). Contained in this package are the files required to connect to Teamcenter using services. Because the sample application you are working with is the Java client, the JAR files must be retrieved from the TEM package and added to the Java project in Eclipse.

   These files are found in the TEM package created by the Business Modeler IDE. If they do not exist, the Business Modeler IDE project must be rebuilt with the correct service targets specified, and then repackaged. These files must be added to the Java project as referenced libraries.

   They are also in the **full_update** folder in the *project-name*_**soa_client_kit.zip** file.

   - **soa_client/jars/TcSoaStrongModel***project***.jar**

     Contains the class definitions for your custom business object types.

   - **soa_client/jars/***prefix***SoaLibStrong.jar**

     Contains the service library stub and the service class.

- **soa_client/types/jars/***prefix***SoaLibTypes.jar**

  Contains the service operations for the service library.

2. Create a custom **RunService.java** class in your Eclipse project to contain a public method to call the custom Teamcenter service.

   The public class **RunService** contains a public method named **runService**.

   There are two main steps required to call any Teamcenter service.

   a. Get the service stub. For example:

   ```
   CustomService c9ServiceStub = CustomService.getService(Session.getConnection());
   ```

   b. Call the service operation from the stub, passing in any required parameters, and receive a return value, if one exists. For example:

   ```
   StringResults valReturn = c9ServiceStub.callC9Validate( theItem );
   ```

   In this example, the service name chosen during creation in the Business Modeler IDE is **Custom**. The Business Modeler IDE appended the word **Service** to the end of the service name. This service operation happens to expect an **Item** object and return a custom data type called **StringResults**.

   > Note:
   >
   > The appropriate imports are added by Eclipse for the **getService** and **callC9Validate** methods. If not, they can be found in the packages contained in the **soa_client** JAR files.

3. Add a call to the **RunService.java** class from the project's main Java class to call the newly created public method to execute the server based code.

   For example, in the **HelloTeamcenter** Java sample application in the **Hello.java** class, add a line below the **Session session ...** line to instantiate the newly written class. For example:

   ```
   RunService c9srv = new RunService();
   ```

   Add a line below the **session.login();** line to call the newly written method. For example:

   ```
   c9srv.runService();
   ```

Basic concepts for using services

Services documentation

The documentation for the different bindings is found in the **soa_client.zip** file in the Teamcenter software distribution image. After you extract the ZIP file, go to the **soa_client/Help.html** file.

The documentation for the different bindings is found in the following directories:

- C++

  soa_client/cpp/docs/

- Java

  soa_client/java/docs/loose/

  soa_client/java/docs/strong/

- .NET

  soa_client/net/docs/

- WSDL

  soa_client/wsdls

Documentation for the services API is also available in the *Services Reference*.

> Note:
>
> The *Services Reference* is available on Support Center.

The *Services Reference* contains the API documentation for the following bindings:

- C++
- Java (loose and strong)
- .NET

Note:

The Persistent Object Manager (POM) and Integration Toolkit (ITK) functions are used by many services.

For documentation about ITK functions, see the *Integration Toolkit Function Reference*.

Siemens Digital Industries Software customization support for Teamcenter Services

Siemens Digital Industries Software is committed to maintaining compatibility between Teamcenter product versions. If you customize functions and methods using published APIs and documented extension points, be assured that the next Teamcenter version will honor these interfaces. On occasion, it may become necessary to make behaviors more usable or to provide improved integrity. Our policy is to notify customers at the time of the release prior to the one that contains a published interface behavior change.

Siemens Digital Industries Software does not support code extensions that use unpublished and undocumented APIs or extension points. All APIs and other extension points are unpublished unless documented in the official set of technical manuals and help files. Any class found in a package or namespace with the name **internal**, such as **Teamcenter::Soa::Internal** or **com.teamcenter.services.internal** are unpublished APIs and are not supported. The Teamcenter license agreements prohibit reverse engineering, including:

- Decompiling Teamcenter object code or bytecode to derive any form of the original source code
- Inspecting header files
- Examining configuration files, database tables, or other artifacts of implementation

Siemens Digital Industries Software does not support code extensions made using source code created from such reverse engineering. If you have a comment or would like to request additional extensibility, contact your Siemens Digital Industries Software customer support representatives.

Note:

Custom service libraries compiled against a major release will not need to be recompiled for minor releases or patches. Binary compatibility for both client and server will be maintained throughout the life of the major release. Recompilation will be required for each subsequent major release.

Service-oriented architecture

Service-oriented architecture, or SOA, refers to a style used in the development of enterprise-class business applications. Traditional business application development has followed a purpose-built paradigm that is now perceived as being inflexible and incapable of evolving at the same rapid pace

that business markets and processes are now changing. SOA provides the flexibility and the opportunity for business applications to keep pace with changes in the global business climate.

The most important characteristics of SOA that allow it to meet these fundamental objectives are:

- Contract-based interfaces

  SOA promotes the use of abstract interfaces independent of underlying business logic implementations. These interfaces represent a contract between clients and the business applications. As long as the client makes a proper request for a service, the server honors it and returns a response in the format specified in the contract. The SOA contract helps to uncouple client and server development activities and allows them to evolve independently so long as the contract remains intact.

- Coarse-grained interfaces

  SOA interfaces also tend to be coarse-grained, in the sense that they ideally represent a complete business transaction. The client makes a request for some type of work to be done; the server executes the full operation without any intervening conversation with the client, and then sends back a fully-complete response to the client. This results in fewer remote calls to API and brings about a nonchatty interface.

- Single entry point to business logic

  After a service is exposed through SOA, it is immediately available to all clients. Because the service is contract-based, the server logic does not require knowledge about the client requesting the service, which allows new client technologies to be adopted without changing the service or invalidating other clients using the service.

Why Teamcenter Services is important

Teamcenter Services provide the underpinnings for high-performance, scalable, WAN-friendly applications. Its contract-based interfaces ensure stability and longevity for applications built using Teamcenter Services.

What you can do with Teamcenter Services

You can use Teamcenter Services to create loosely coupled integrations with Teamcenter, regardless of the programming language standards and operating environments prevalent in your organization. You can use Teamcenter Services to integrate Teamcenter with the programming language of your choice. Teamcenter Services include client libraries for Java, C++, C# (.NET), as well as WS-I compliant WSDL. Services can also be used both for interactive, end-user integrations as well as noninteractive, system-to-system integrations.

Teamcenter Services advantages for your organization

Teamcenter Services provide service interfaces that are both language and technology independent. All applications can use the same interfaces to backend systems, resulting in lower system and maintenance costs for your organization. The coarse-grained nature of the Teamcenter Services interfaces allows work to be done in a single request-response cycle, making application performance over high-latency networks possible.

Integration with Teamcenter Services is best achieved with the provided integration libraries. These libraries remove the burden of communication handling and data model management and allow integrators to focus solely on using the exposed capability. The libraries also provide a client-side data model (CDM) with built-in support for caching, SSO, and FMS (that provides the underlying requirements to communicate with the server). Application developers can concentrate on the functionality that helps the business rather than on low-level infrastructure development.

If environment or deployment issues limit or restrict the usage of the provided libraries, the inclusion of industry standard WSDL Web service interfaces (that are compliant with the WS-I Basic Profile) enables a generic integration with supplier and partner systems. WSDL integration requires integrators to:

- Create and maintain the service stubs and proxies.
- Manage sessions (HTTP cookies).
- Integrate with the SSO and FMS servers.
- Manage the client data model.

The WSDL does not define XSD complex types for all business model types defined in Teamcenter; instead, a single XSD complex type represents any business model type. Each WSDL defined service is independent, so business model types returned from each service request create new instances of those objects on the client, even though they represent the same instance of the object on the server.

Teamcenter Services framework

Introduction to the Teamcenter Services framework

The Teamcenter Services framework consists of:

- Autogenerated service skeletons on the Teamcenter server.
- Autogenerated service stubs in each of the supported client technologies.
- Standard HTTP communication pipelines and Web-standard middleware.
- A CORBA-based communication pipeline for use in certain Teamcenter configurations.

- An autogenerated client data model that exposes the Teamcenter server data model to service-based client applications.

The actual Teamcenter service implementations are an integral part of the Teamcenter Server business logic.

Available message transports

Client applications using Teamcenter Services can be configured to take advantage of different communication protocols and message transports to connect to the Teamcenter server.

- In a two-tier Teamcenter configuration, an instance of the Teamcenter server process executes on the same end-user computer as the client application, and the client connects to the server by using TCCS. Purely C++ clients also have the option of connecting directly using **IIOP**.

- In four-tier Teamcenter configurations, the client and server processes are typically separated by a network, and there is a wider variety of communication styles available. All the supported client languages can be used (C++, Java, .NET, or WSDL) by using the TCCS module or standard HTTP communication networks.

  The TCCS module connects to the Teamcenter server over **HTTP(S)**, but supports a wider variety of deployment configurations than does the direct **HTTP** connection.

Common message structures (input and output)

Teamcenter Services are implemented using a small set of patterns or characteristics to ensure consistency in how the operation signatures look to developers, as well as to ensure adherence to internal best practices for Teamcenter. In general, service signatures are set-based and broadly useful over a range of applications.

Set-based services can be seen in operation signatures that take as input an array or vector of structures. This allows the client application developer to request a similar operation on multiple data model items with a single call to the server.

Being broadly applicable is the second noticeable pattern in the way operations are implemented. For example, even though the attributes necessary to create different types of business items can vary widely, the mechanics of doing so are very similar from an end-user standpoint. A single Teamcenter service can usually be applied to multiple business object types. As a result, the overall set of operations that a client developer must learn is greatly reduced. This pattern also provides a clean mechanism for extending the type of business objects that can be acted upon without having to introduce new operations with slightly different signatures for each one.

While set orientation and broad applicability are the default approaches to creating service operations, there are instances where explicit operations on some business items are provided for convenience or common usage. This approach provides a balance between generalized operations and the need for developer convenience.

Request and response interaction patterns

A typical request/response cycle used with Teamcenter Services requires the following steps:

1. Populate a set of structures representing the business items that you want to request an operation on.

2. Gather the populated structures into a vector or array as appropriate for the client technology you are using (C++, Java, .NET, and so on).

3. Instantiate a service from the service stub.

4. Call the operation you are interested in having the Teamcenter server execute.

5. Wait for the Teamcenter server to respond to your request.

6. React to the server response as appropriate for your application. Typically, you must listen for updates from the client data model **ModelManager** component and update the application's user interface to reflect the results of the operation.

Client data model

The language-specific services client libraries contain a client data model (CDM) component that acts as a local datastore for Teamcenter data returned by service method calls.

If the client application uses loosely typed client libraries, the client data model is populated with instances of the model object (**ModelObject**), and a list of a name/value pairs representing the object's properties. If the client is using the strongly typed client libraries, the client data model is populated with typed objects that correspond one-to-one with their server-side object type. If the specific server-side type does not exist on the client, the nearest available parent type is constructed. All properties associated with the type are available to the client application.

Client applications can access data in the CDM:

- Through model manager (**ModelManager**) listeners.

- Directly from the service data (**ServiceData**) object.

- Through the **ClientDataModel** class.

> Note:
>
> - If a Teamcenter feature is installed which adds properties to a foundation business object, the added properties will not have their own getter methods. The base **ModelObject.getProperty(propertyName)** method can be used to get any property values that are defined in child templates. The Aerospace and Defense solution is one example of this.

- When working with object types, consider the following:
  - The **.getProperty("object_type")** method returns the localized display name.
  - To retrieve the nonlocalized (database) name of the type, use the **.getTypeObject.getName()** method.
  - When comparing or checking object type, we recommend that you use the database name instead of the display name. This avoids the need to compensate for localization.

Java bindings of the client data model

The Java client bindings of the Team Services offer three client data models: loose, strong, and **TCComponent**.

When developing a client application, you must choose which Java client binding to use.

Loose

This client data model is loosely coupled to the server-side data model, where there is one class (**ModelObject**) that represents all server-defined business object types. The **ModelObject** class has a generic **getPropertObject()** method to retrieve any named property value from the object.

```
ModelObject.getPropertyObject("data_released")
```

This CDM has the smallest footprint, but uses the generic property access method which would not catch any property typos until runtime.

Strong

This client data model is strongly coupled to the server-side data model, where there is an explicit class defined in the client data model for each business object type defined on the server. For example, **ItemRevision**, **BOMLine**, and so on. These classes have a getter method for each property defined for the business object type.

```
ItemRevision.get_date_released()
```

This CDM has a largest footprint, but provides type safety through explicit classes and property getter methods.

TCComponent

This client data model is similar to the strong client data model, but is exclusive to the rich client. There is a **TCComponent***xxx* class for each business object type defined. For example, **TCComponentItemRevision**, **TCComponentBOMLine**, and so on. However, unlike the strong client data model, the **TCComponent** client data model is not autogenerated from the server-side data model, but rather manually created by Siemens Digital Industries Software developers as needed. Therefore, there is not a one-to-one mapping of object types to TCComponents. Most business object types have a corresponding **TCComponent** class, but not all.

This CDM provides the same type safe explicit property methods as the strong binding, but because the **TCComponent** class definitions are already included in the rich client, does not incur the large footprint penalty.

Errors and exceptions

Set-based operations use business objects and associated parameters to control how the Teamcenter server processes each object with the following results:

- Success

  A successful operation typically results in the return of a set of object references in the service data (**ServiceData**) object and the object properties in operation-specific structures.

- Failure of the requested actions on one or more of the input objects

  When an operation fails on one or more of the input objects, but succeeds on others, the failures are referred to as *partial errors*. In this case, the service data object contains appropriate **ErrorStack** entries for each failed item or operation.

- Failure

  When an operation fails, a service exception is generated and returned to the calling client and must be handled by the application logic that called the service operation.

Object property policies

The object property policy defines which properties are included for a given object type when returning objects to the client. The policy is used by the Teamcenter Services framework to automatically and uniformly inflate the properties of objects contained in the service data object of the returned data structure. A Teamcenter server installation is deployed with a default policy plus any number of specific policies defined by the different client applications. Each installed application using Teamcenter Services typically has one or more policies installed on the server.

Basic tasks for using Teamcenter services

Calling Teamcenter Services

The basic pattern for calling Teamcenter Services operations is as follows:

1. Populate a set of structures representing the business items on which you want to request an operation.

2. Gather the populated structures into a vector or array as appropriate for the client technology you are using (C++, Java, .NET, and so on).

3. Request a service stub from the **Connection** object of your session.

4. Call the service operation.

5. Wait for the Teamcenter server to respond to your request.

6. React to the server response as appropriate for your application. Typically this means listening for updates from the CDM manager and updating the application's user interface to reflect the results of the operation.

Most operations use arrays of structures as input, and return a standard service data object along with one or more operation-specific output structures. Refer to the online services references or the WSDL files and schemas for details about these structures.

Configure Teamcenter services

Connecting a client application to the server

The Teamcenter server does not require any additional configuration to allow your client application to use Teamcenter Services. However, your client application must be configured to connect to the Teamcenter server. The Teamcenter server can be deployed in a two-tier or four-tier environment. Use the **Connection** class to determine how your client application will communicate with the Teamcenter server.

- `com.teamcenter.soa.client.Connection`
- `Teamcenter::Soa::Client::Connection`

For WSDL-based clients, the SOAP toolkit that is used determines how the server address is configured.

- Two-tier or four-tier TCCS communication

  The Teamcenter client communication system (TCCS) supports both two- and four-tier deployments. The TCCS module provides for a more robust support of proxy servers than direct **HTTP** communication. The **protocol** and **hostPath** arguments on the **Connection** class constructor configure the client application for TCCS communication. The TCCS environment itself dictates if it uses two- or four-tier communication.

protocol	hostPath
TCCS	Identifies the TCCS environment to use, and uses the following syntax: `tccs://`*environment* • *environment*

protocol	hostPath
	Specifies the TCCS environment name.

- Two-tier IIOP communication (C++ client only)

  The **protocol** and **hostPath** arguments on the **Connection** class constructor configure the client application for the two-tier environment.

protocol	hostPath
IIOP	Identifies the port defined by the CORBA ORB endpoint, and uses the following syntax: `iiop:`*host*`:`*port*`/`*server-id* • *host* Specifies the host machine the Teamcenter server is executing on (**localhost** is allowed). • *port* Specifies the port number on the Teamcenter machine configured for communications. • *server-id* Specifies the CORBA ID of the Teamcenter server.

- Four-tier direct HTTP communication

  The **protocol** and **hostPath** arguments on the **Connection** class constructor configure the client application for the four-tier environment.

protocol	hostPath
HTTP	Defines the Teamcenter Web tier address, and uses the following syntax: `http//:`*host*`:`*port*`/`*app-name* • *host* Specifies the network name of the Teamcenter Web tier host machine.

protocol	hostPath
	• *port* Specifies the port number on the Web tier machine configured for communications. This is the port number on the Web application server. For example, if you are using Apache Tomcat, the default value is **8080**. • *app-name* Specifies the application name of the deployed Teamcenter Web tier processes. The default value is **tc** but may have changed during the installation and configuration of the Teamcenter server software.

- Four-tier HTTP communication using WSDL stubs

  The SOAP toolkit you choose determines how communication with the Teamcenter sever is configured. However, each SOAP toolkit requires a URL for each service. The URL uses the following syntax:

  `http://`*host*`:`*port*`/`*app-name*`/services/`*service-port*`?wsdl`

  Or:

  `http://`*host*`:`*port*`/`*app-name*`/services/`*service-port*

  Where:

  - *host*

    Specifies the network name of the Teamcenter Web tier host machine.

  - *port*

    Specifies the port number on the Web tier machine configured for communications. This is the port number on the Web application server. For example, if you are using Apache Tomcat, the default value is **8080**.

  - *app-name*

    Specifies the application name of the deployed Teamcenter Web tier processes. The default value is **tc** but may have been changed during the installation and configuration of the Teamcenter server software.

  - *service-port*

Specifies the name of the Teamcenter Services operation being called.

- *wsdl*

  Specifies the optional URL **wsdl** query string to return the WSDL service. The URL without the query string executes the service request.

Client libraries

Client libraries organization

The Teamcenter Services client libraries are located on the Teamcenter software distribution image in the **soa_client.zip** file. This file includes the actual Teamcenter Services client libraries (Java, C++, and .NET), any libraries on which they depend, and API documentation.

To get started using services, after you extract the ZIP file, go to **soa_client/Help.html**.

The client libraries for each language or technology are organized as follows:

- Java

  - **TcSoa***library-name***Types***release-version***.jar**

    Contains the XSD bindings for the service data structures.

  - **TcSoa***library-name***Strong***release-version***.jar**

    Contains the strong model bindings of the service stubs.

  - **TcSoa***library-name***Loose***release-version***.jar**

    Contains the loose model bindings of the service stubs.

  All Teamcenter Services JAR files are Eclipse RCP-ready plug-ins, and may be referenced in a standard Java JVM classpath or as a plug-in to an Eclipse RCP application.

- C++

  - **libtcsoa***library-name***types.dll**

    Contains the XSD bindings for the service data structures.

  - **libtcsoa***library-name***strongmngd.dll**

    Contains the managed strong model bindings of the service stubs.

- **libtcsoa***library-name***strong.dll**

  Contains the strong model bindings of the service stubs.

  > Warning:
  >
  > The **strong.dll** bindings are deprecated in favor of the **strongmngd.dll** bindings. Adding improved memory management to the C++ bindings in Teamcenter 8 necessitated changing method signatures in the client bindings to the **strongmngd.dll** structure. For all future C++ strong bindings usage, implement the **strongmngd.dll** bindings.

- C# (.NET)

  - **TcSoa***library-name***Types.dll**

    Contains the XSD bindings for the service data structures.

  - **TcSoa***library-name***Strong.dll**

    Contains the strong model bindings of the service stubs.

Client library dependencies

The Teamcenter Services client libraries ship with all required dependent libraries. Remember to include the **libs** directory for your chosen programming language into your linker configuration.

Deprecation of C++ strong client libraries

In the C++ bindings, the **strong.dll** bindings are deprecated in favor of the **strongmngd.dll** bindings. Adding improved memory management to the C++ bindings in Teamcenter 8 necessitated changing method signatures in the client bindings to the **strongmngd.dll** structure. For all future C++ strong bindings usage, implement the **strongmngd.dll** bindings.

In Teamcenter 2007.1, management of the Client Data Model in the C++ client was limited with respect to removing objects from the store. All model object (**ModelObject**) construction and destruction is done in the **ModelManager** class. As data is returned from service operations, the model manager constructs instances of model objects and adds them to an internally managed store. This store continues to grow over the life of the session. The model manager exposes methods to remove objects form this store:

```
ModelManager::removeObjectsFromStore
     ModelManager::removeAllObjectsFromStore
```

However, the implementation of these methods blindly deletes the named model object instances, without any regard to the client application still holding pointers to those objects. This limitation leaves the client application two choices:

- Allow the Client Data Model store to grow throughout the session and use up valuable memory.
- Remove objects form the Client Data Model store and risk dangling pointers accessing invalid memory. Garbage collection makes this a nonissue in the Java and .NET client bindings.

To address this limitation, reference counting is added to the Client Data Model in Teamcenter 8. This architecture change requires a change in the vast majority of the method signatures in the client framework and service interfaces. To ease the transition of client applications to the new managed memory feature, the existing libraries are deprecated as a whole, and new libraries are introduced.

Deprecated	Replacement
soa_client/cpp/includes/cpp/include (formerly **soa_client/cpp/include**)	**soa_client/cpp/includes/cpp-managed/include**
libtcsoaclient.dll	**libtcsoaclientmngd.dll**
libtcsoa*library-name***strong.dll**	**libtcsoa***library-name***strongmngd.dll**

If you have existing client applications, you may continue to use the existing libraries, but you should consider migrating to the new set of libraries soon. If you choose to continue using the deprecated libraries, at a minimum, you must change your compile options to retrieve the header files from the new location in the **soa_client** kit:

```
/I soa-client-root\cpp\includes\cpp\include
```

In Teamcenter 2007.1, or the deprecated libraries, all references to the **ModelObject** object is through raw pointers. In Teamcenter 8, this is replaced with the **AutoPtr<>** template class that includes reference counting of the underlying raw pointer. This ensures that the instance is not deleted until all references to the pointer are released. The client framework and service interfaces have changes similar to this:

```
Existing:    ModelObject *        ModelManager::getObject( string uid );
New:    AutoPtr<ModelObject> ModelManager::getObject( string uid );
```

To take advantage of the reference counting of the **AutoPtr** class, you must make similar changes to your client application code. This involves three basic changes:

1. Modify your compiler options to use this **include** folder:

   ```
   /I soa-client-root\cpp\includes\cpp-managed\include
   ```

2. Modify your link options to link against the new set of libraries and add **mngd** to the end of existing library names. The **libtcsoacommon.dll** and **libtcsoa***library-name***types.dll** libraries are shared between the old set and new set of client libraries.

3. Compile your code. The compiler returns errors for all code that needs to change. You must make the following types of code changes. Apply these changes to the base **ModelObject** class and all other classes in the Client Data Model (**ItemRevision**, **BOMLine**, and so on).

   - Variable declaration

     - Old way

```
// Declare pointers for an Item, Form and vector of ModelObjects
// initialize to NULL
Item  *theItem   = NULL;
Form  *theForm   = NULL;
vector< ModelObject *> objects;

...

// Use the Item
theForm = theItem->get_item_master_tag();
objects = theItem->get_revision_list();
```

     - New way

```
// Declare AutoPtr for an Item, Form and vector of ModelObjects
// all AutoPtrs are instantiated as NULL
Teamcenter::Soa::Common::AutoPtr<Item>      theItem;
Teamcenter::Soa::Common::AutoPtr<Form>      theForm;
vector< Teamcenter::Soa::Common::AutoPtr <ModelObject> > objects;

...


// Use of the ModelObject variables remains unchanged
theForm = theItem->get_item_master_tag();
objects = theItem->get_revision_list();
```

   - Casting

     - Old way

```
// Declare the variables
User        *user;
ModelObject *anObj;
Folder      *folder;

user = sessionService->login(name,pass, “”,””, descrim ).user;

// Cast to the base ModelObject class
anObj = (ModelObject*)user;
```

```
// Cast to a speicfic sub-type, if not NULL
// do something with the new pointer
user = dynamic_cast<User*>(anObj);
if(user != NULL)
{
  folder = user->get_home_folder();
}
```

- New way

```
// Declare the variables
Teamcenter::Soa::Common::AutoPtr<User>        user;
Teamcenter::Soa::Common::AutoPtr<ModelObject> anObj;
Teamcenter::Soa::Common::AutoPtr<Folder>      folder;

user = sessionService->login(name,pass, “”,””, descrim ).user;

// Cast to the base ModelObject class
anObj = user.cast<ModelObject>();

// Put the cast to a speicfic sub-type, in a try/catch block
// if std::bad_cast not thrown, then cast was successful
try
{
user  = anObj.dyn_cast< User >();
folder = user->get_home_folder();
}
catch(std::bad_cast&){}
```

- NULL test and NULL assignment

  The test for NULL and NULL assignment of a **ModelObject** object can remain unchanged when upgrading to the **AutoPtr** version. However, this syntax can be misleading when working with **AutoPtr** objects. The following replacement code shows equivalent functionality that is more natural to the **AutoPtr** version:

  - Old way

```
// Declare the variables
LoginResponse   response;
User          * user;

// Call a service that returns a ModelObject (User)
response = sessionService->login(name,pass, “”,””, descrim );
user = response.user;

// Test that instnace against NULL
```

```
if( user == NULL)
{
   ...
}
else
{
    // Assign NULL to the pointer
  user = NULL;
}
```

- New way

```
// Declare the variables
// The service data structures do not change, only
// references to  ModelObjects
LoginResponse                          response;
Teamcenter::Soa::Common::AutoPtr<User> user;

// Call a service that returns a ModelObject (User)
response = sessionService->login(name,pass, “”,””, descrim );
user = response.user;

// Since we are not dealing directly with pointers,
// NULL does not make sense here, use the isNull method from the
// AutoPtr template class.
if( user.isNull())
{
   ...
}
else
{
    // Release the instance and
  user.release();
}
```

Obsolete Java client data model methods

To fully integrate the SOA framework and services into the Teamcenter rich client, the client data model classes/interfaces defined in the rich client and SOA must be merged. In the rich client and SOA client data models, there are two sets of interfaces that have conflicts:

- SOA

  com.teamcenter.soa.client.model.ModelObject
  com.teamcenter.soa.client.model.Property

- Rich client

com.teamcenter.rac.aif.kernel.InterfaceAIFComponent
com.teamcenter.rac.kernel.TCProperty

The following table summarizes the method name changes. Calling code that references the classes/interfaces old methods must be changed to reference the new methods as noted in the table. There is no logical or functional change; only the method names are changed.

Old methods	New methods
ModelObject.getProperty	**ModelObject.getPropertyObject**
ModelObject.getType	**ModelObject.getTypeObject**
Property.getDateValue	**Property.getCalendarValue**
Property.getDateArrayValue	**Property.getCalendarArrayValue**

Note:

This is not a deprecation of classes/methods but an outright change of method names. This results in calling code in SOA client applications failing to compile until that source code is modified to account for these changes.

Retrieving basic item information

Basic information about items returned from an operation is contained within the service data object returned from the call. The information in the service data (**ServiceData**) object is controlled through the **ObjectPropertyPolicy** object in place at the time of the operation request.

If additional information about one or more of the returned objects is needed, the **DataManagement.getProperties()** operation can be called.

The input to the **DataManagement.getProperties()** operation is a vector or array of model object (**ModelObject**) objects from the CDM and a second array or vector containing the names of the Teamcenter properties to retrieve for each model object in the first array, as in:

```
getProperties(ModelObject[ ] objects, String[ ] attributes)
```

The return structure is a standard service data structure that contains the requested model objects with the requested attributes populated into them in addition to the attributes specified in the current **ObjectPropertyPolicy** object. Returning both sets of attributes make it possible for the client application to use the returned model objects directly without worrying about merging the new attributes into the previously returned default set.

Note:

There are performance impacts pertaining to this operation. Attributes used by the client application should be requested on an as-needed basis rather than prefetching a large list of attributes in case the application may need them later. This is the trade-off between UI responsiveness and performance. The balance between them often depends on the user's expectations when using a specific client application.

Uploading and downloading persistent files

While much of the information managed within Teamcenter is considered *metadata* (data about items), it is often useful to associate bulk data with the metadata. This bulk data typically consists of various files containing product design geometry from a CAD application, specifications, analytical reports, and so on. References to these files are stored in Teamcenter in a construct called a dataset that is associated with an item in the Teamcenter database. The actual file content is persisted within Teamcenter File Management System.

Uploading a file and attaching it to a dataset is a three-step process:

1. Request a ticket from Teamcenter File Management System (FMS).

2. Use the ticket to request FMS to upload and store the file.

3. Commit the file into the appropriate dataset using the **commitDatasetFiles()** operation.

An FMS ticket is a token that grants the holder permission to carry out a certain operation, usually an upload or download of one or more files. The ticket generation process interfaces with the Teamcenter access management system to ensure that only authorized users can read, write, or modify bulk data and datasets using FMS.

For files referenced by datasets, the client application can call either:

```
getDatasetWriteTickets(FileManagement.GetDatasetWriteTicketsInputData[ ] inputs)
```

or:

```
getFileReadTickets(ImanFile[ ] files)
```

depending on whether it needs to perform a read or a write operation. Both operations are set-based, which allows multiple tickets to be requested at a single time.

On a write operation, the uploaded files can be attached to datasets and committed to the Teamcenter database using:

```
commitDatasetFiles(FileManagement.CommitDatasetFileInfo[] commitInput)
```

Importing and exporting data

Often a client application wants to upload a local file and process its content into the Teamcenter database or extract some information from Teamcenter into a file and download the file for local use. The upload operation is called an *import* and the download operation is called an *export*. In either case, the file can be considered a container that Teamcenter does not have to persist or track once its contents are transferred into or out of the system. These are referred to as *transient* files. Similarly to persistent files, transient files are transferred using File Management System (FMS).

To import data into Teamcenter using one or more transient files, the client application must first obtain the appropriate write tickets from Teamcenter. The client application can pass the ticket to FMS with the local file names and the name to import the files as.

After the files are uploaded, the same tickets can be used to call the Teamcenter operation that retrieves the files from FMS, and to perform the tasks necessary to parse the transient file contents and process them into the Teamcenter database. After the import is complete, FMS by default deletes the transient file, although the client application can request that the file not be deleted by setting a flag on the ticket request.

The sequence of operations that is used to import data into Teamcenter is:

```
getTransientFileTickets(someInputStructure[ ] inputs)
```

followed by a request to FMS to upload the files, and completed with:

yourImportProcessor(*yourInputStructure*`[ ] inputs)`

Using transient files and FMS to export information from Teamcenter to a local file is conceptually similar to the import use case, but it employs a slightly different sequence. The first step in an export operation is to call a Teamcenter service, which performs the data extraction, requests the appropriate FMS transient file tickets, writes the files to FMS, and returns the FMS tickets to the calling client.

With the returned tickets, the client application can request FMS to download the files. After the download completes, the client application can process the files in whatever manner it chooses. By default, the FMS deletes the transient file from the Teamcenter volume. There is an option on the **getTransientFileTickets()** operation to request that the file not be deleted from the transient volume.

2. Teamcenter Services framework components

Establishing a Teamcenter session

Process for establishing a Teamcenter session

There are several steps to establishing a Teamcenter session for your client application:

1. Instantiate a **CredentialManager** object.
2. Instantiate a **Connection** object.
3. Call the **SessionService.login()** operation.
4. Handle exceptions.

CredentialManager interface

The first step in establishing a Teamcenter session is to instantiate a credential manager (**CredentialManager**) object. The Teamcenter Services framework includes a credential manager interface for which the client application developer must create an implementation. The developer must decide how best to implement the credential manager interface in the context of the application. The credential manager implementation may cache credentials from the user, typically user name and password or a Teamcenter SSO token. This allows for silent reauthentication in the case of dropped communications. Depending on security constraints within your organization, the credential manager implementation can prompt the user for credentials if there is a session validation issue.

Your client application session may terminate because of inactivity that lasts longer than the maximum time set by the site administrator or because of a communications network failure. When this occurs, the client application must reauthenticate. Instead of having the client application catch and react to an **InvalidUser** exception for every service request, the Teamcenter Services framework does this automatically. When the Teamcenter server does return an **InvalidUser** exception, the client framework uses the credential manager interface in the **Connection** object to get the user's credentials and send the **SessionService.login()** service request to re-establish a session. Once validated, the service request that originally caused the **InvalidUser** exception is resubmitted automatically by the Teamcenter Services framework.

Connection object

The connection (**Connection**) object manages a connection for a single user to a single server. In the most basic case of a client application having a single user connecting to a single server, the client application must instantiate and maintain a single instance of the connection object. This connection

object is used to instantiate the appropriate service stubs as service operations are invoked by the client application.

Even though the connection object is instantiated, no communication is made on the configured connection until a Teamcenter Services operation is actually invoked.

A client application can support multiple sessions by using multiple instances of the connection object. Each instance of the connection object maintains a separate session with the Teamcenter server (provided a unique user/discriminator combination is used on the logon) and a separate client data model. Each service request is invoked with an instance of the connection object, so it is the client application's responsibility to manage the different **Connection** object instances.

Each time the server is accessed, a new connection object is created. This process can be optimized by serializing the connection object. When the first connection object is created, all the parameters and context information in this connection object are converted to strings and stored in memory or a file in the user directory. This serialized connection object is issued for subsequent access to the server. This is useful in re-establishing the previous connection state.

The following connection object values are serialized:

- User ID
- Arguments used to construct a connection object (server path, binding style, and so on)
- Connection options (any option set through the **Connection.setOption** method)
- Client state (all the values cached in the **SessionManager** and passed in the RCC headers). For subsequent connections, the serialized client context map is only applied for the matching user ID.

The serialized object is stored in a text file that is saved in the user directory after running it through a cryptographic hash function (for example, MD5). This is done for security purposes. The saved file is readable by all client bindings. A connection object can be instantiated from this serialized form. The client state variables are applied after the user has logged on with the matching user ID.

The following options can be set for the connection object using the **Connection.setOption** method:

- **OPT_CACHE_MODEL_OBJECTS**

  By default, the **OPT_CACHE_MODEL_OBJECTS** option in the **Connection.setOption** method is set to **true**, instructing the model manager to keep a cache of all returned model objects. To reduce memory usage, set this option to **false**. This instructs the model manager to process service responses without keeping a cache of returned model objects; when the returned data structure is no longer needed, the model object instances go to garbage collection. This only impacts you if you write your own clients.

  A cache of model schema is created for each server address rather than for each connection. Multiple connections that are connected to the same server share the model schema.

- **OPT_SHARED_DISCRIMINATOR**

  The **OPT_SHARED_DISCRIMINATOR** option controls whether four-tier clients use a specific discriminator that ensures they are connected to the same server as other SOA clients running on this machine. Only clients logging on with the same user are able to make use of server sharing. In a custom Teamcenter Services client application, this sharing is disabled by default. Set this before logon for client applications running in a four-tier installation. Two-tier clients usually share a server without special setup.

  > Note:
  >
  > Enabling the **OPT_SHARED_DISCRIMINATOR** option automatically enables the **OPT_SHARED_EVENTS** option.

- **OPT_SHARED_EVENTS**

  The **OPT_SHARED_EVENTS** option controls whether this client participates in session sharing with other clients on this machine. Enabling this flag allows the application to keep in synchronization with other clients sharing the server through use of shared session events and model events. In a custom Teamcenter Services client application, this sharing is disabled by default. To enable this sharing, set the **OPT_SHARED_EVENTS** parameter on the connection object to **true** before initiating a logon.

Logging on to Teamcenter

Before any service operation can be invoked, the user must be authenticated. This is done through the **SessionService.login** operation.

The Teamcenter architecture varies from other client server architectures in that there is a dedicated instance of the Teamcenter server per client application. However, there are use cases where it is desirable for a single user to have multiple applications running with each sharing a single instance of a Teamcenter server. This is controlled through the following arguments:

- **hostPath**

  Specifies the host for the **Connection** class constructor.

- **username**

  Specifies the user name input to the **SessionService.login** operation.

- **sessionDiscriminator**

  Specifies the session input to the **SessionService.login** operation.

The **hostPath** argument determines the server machine that the client connects to. Once there, the pool manager on that host uses the **username** and **sessionDiscriminator** arguments of the logon request

to determine which Teamcenter server instance to assign the client to. If the pool manager has an existing Teamcenter server instance with the **username/sessionDiscriminator** key, the client is assigned to that existing instance of the Teamcenter server (therefore sharing the server with another client). Otherwise, a new instance of the Teamcenter server is used. There are a few general scenarios for the **sessionDiscriminator** argument:

- Blank

  If the user **jdoe** logs on to Teamcenter using two or more client applications using a blank **sessionDiscriminator** argument (for example, **jdoe/**), all of those clients are assigned to the same Teamcenter server instance. These client applications can be running on the same or different client hosts.

- Constant

  If the user **jdoe** logs on to a client application (for example, **jdoe/myApp**), this is similar to the blank **sessionDiscriminator** argument. The difference is that only multiple instances of the client application using **myApp** started by **jdoe** share the same Teamcenter server instance.

- Unique

  If the user **jdoe** logs on using a unique random-generated string (for example, **jdoe/akdk938lakc**), the **sessionDiscriminator** argument ensures the client application has a dedicated instance of the Teamcenter server.

The scenario you use depends on how your client application is used in the integrated environment. The most common case is the unique **sessionDiscriminator** value.

A client session is a unique instance of the **Connection** object. In most cases, each client application manages a single instance of the **Connection** class, but it is fully supported for a single application to manage multiple instances of the **Connection** class. For server assignment, it does not matter if the **SessionService.login** requests come from multiple client applications or a single client.

Whenever possible, use the **SessionService.logout** operation to explicitly log off from the Teamcenter server and to terminate the server instance. This is preferable to allowing the session to time out due to inactivity. Waiting for the server to time out can cause server resources to be held unnecessarily. If your client application chooses to share a Teamcenter server instance, the server does not terminate until the last client sends the **SessionService.logout** request.

Using Security Services

When the Teamcenter server is configured for single sign-on (**SSO**) authentication using Security Services, the client application must still log on to Teamcenter (call the **SessionService.login** service operation), but the user is not prompted for credentials. When in **SSO** mode, instead of the client application providing an implementation of the **CreditialManager** interface, the client application uses the **SsoCreditials** class (which is part of the **SOA** client libraries) as the credential manager. The

SsoCredentials class calls the appropriate **SSO** client to obtain the user name and security token. These credentials are used as input to the **SessionService.loginSso** service operation.

ExceptionHandler interface

Similar to the **InvalidUser** exception handled by the credential manager interface, any service request can also throw an **InternalServer** exception. This exception is most commonly caused by a configuration, communication, or programming error. To alleviate the need for the client application to catch this exception with every service request, your client application can implement the **ExceptionHandler** interface and add it to the **Connection** object.

Your client application's implementation of the **ExceptionHandler** interface can determine how these exceptions are handled. The handler can display the error message, prompt the user to fix the problem and send the request again, throw a **RunTimeException** exception, or exit the application.

Code example

The following code snippets put together the four steps for establishing a session with the Teamcenter server:

Instantiate the correct CredentialManager

Choose between **SSO** mode or standard credentials provided by your application.

```
CredentialManager credentialMgr;
    if( ssoMode )
        credentialManager = new SsoCredentials(ssoServerUrl, ssoAppID);
    else
        credentialManager = new YourAppCredentialManager();
```

Create the Connection object

```
Connection connection = new Connection("http://abc.com:7001/tc",
                        credentialMgr, SoaConstants.REST, SoaConstants.HTTP);
ExceptionHandler  expHandler = new YourAppExceptionHandler();
connection.setExceptionHandler( expHandler );
```

Get the service stubs

```
SessionService sessionService =
               SessionService.getService(connection);
SavedQueryService queryService =
                  SavedQueryService.getService(connection);
```

Get credentials

```
    // Through your application's CredentialManager
// or through the SSO client.
```

```
InvalidUserException dummyExp = new InvalidUserException();

String[] credentials = credentialManager.getCredentials( dummyExp );
String name   = credentials[0];
String pass   = credentials[1];
String group  = credentials[2];
String role   = credentials[3];

boolean loggedIn = false;
while( !loggedIn )
{
     try
     {
          // Login to
    Teamcenter
          LoginResponse
    loginResp;
          if(ssoMode)
            loginResp =
  sessionService.loginSso(name, pass, group, role,”en-US” ”uniqueId”);
          else
            loginResp =
  sessionService.login(name, pass, group, role,”en-US” ”uniqueId”);
          loggedIn =
    true;
     }
     catch
  (InvalidCredentialsException ice)
     {
        credentials =
  credentialManager.getCredentials( ice );
        name   = credentials[0];
        pass   = credentials[1];
        group  = credentials[2];
        role   = credentials[3];
     }
}

// Once logged in, call any other service operation
GetSavedQueriesResponse savedQuiries = service.getSavedQueries();
```

Calling services

Invoking a Teamcenter service from the available client bindings is generally the same and only differs in language syntax. Services are accessed through an abstract class or interface for a *base service*. The actual service request goes through a generated stub class which takes care of formatting the request and sending it to the Teamcenter server over the specified transport protocol. The client application is responsible for obtaining the correct stub implementation through the static **getService()** method on the base service class using information from the **Connection** object. As long as the run-time stub is instantiated by the Teamcenter Services framework, the application code can remain neutral to the actual transport protocols and binding styles configured. Following are examples of instantiating a base service and invoking an operation in three different language/model bindings, with bold text indicating the differences between the language/model bindings.

- Java strongly typed data model:

```
import com.teamcenter.services.strong.core.DataManagementService;

// Instantiate the service stub
DataManagementService service =  DataManagementService.getService( connection );

// Invoke the service
CreateFoldersResponse out = service.createFolders( folders, container, "child" );
```

- Java loosely typed data model:

```
import com.teamcenter.services.loose.core.DataManagementService;

// Instantiate the service stub
DataManagementService service =  DataManagementService.getService( connection );

// Invoke the service
CreateFoldersResponse out = service.createFolders( folders, container, "child" );
```

- C++ strongly typed data model:

```
include <teamcenter/services/core/DataManagementService.hxx>

// Instantiate the service stub
DataManagementService* service = DataManagementService.getService( connection );

// Invoke the service
CreateFoldersResponse out = service->createFolders( folders, container, "child");
```

- .NET strongly typed data model:

```
using Teamcenter.Services.Strong.Core;

// Instantiate the service stub
DataManagementService service =  DataManagementService.GetService( connection );

// Invoke the service
CreateFoldersResponse out = service.CreateFolders( folders, container, "child" );
```

Client data model components

The key components of the client data model (CDM) are the **ModelObject**, **ServiceData**, **ModelManager**, **ClientMetaModel**, and **ClientDataModel** classes in the **com.teamcenter.soa.client.model** package.

In general, a client application should *not* be creating or instantiating business objects (**ModelObjects**) like **ItemRevision**, **BOMLine**, and so on. These object instances are returned from service operation calls. If you are trying to create a brand new instance of a given business object type, you *must* use a service operation call like `DataManagement.createObjects`.

There are some edge cases where the client application has the UID of the desired object but needs a full instance of that business object. In this case, the developer could use the `ClientDataModel.constructObject` method. This will create the object instance, but it will

not have any property values. The primary use case for creating an object this way is that a **ModelObject** is required as the input of a service operation and the client only has the UID available. To construct a **ModelObject** instance of an existing business object with property values, use the `DataManagement.loadObjects` service operation.

The **ServiceData** class is a common data structure used to return sets of business objects from a service operation response. This structure holds lists of **ModelObject** objects that are organized by the action taken on them by the service operation: **Created**, **Updated**, **Child-Change** (rich client only), **Deleted**, or **Plain** (objects that the service returns when no changes have been made to the database object). Client applications can directly access the four lists using class methods. Methods are available to return the number of object references in each list as well as to return the object at a specific index within the list, as shown in the following example:

```
ServiceData serviceData = fooServicep.operationBar(...);
// Loop through one or more of the arrays contained within ServiceData.
// Service documentation should make it clear which arrays may have data
for(int i=0; i< serviceData.sizeOfCreatedObjects();i++)
{
    ModelObject singleItem = serviceData.getCreatedObject(i);

    // Since we are working with the Strongly Typed Data Model
    // we can cast the singleItem object to something more specific
    if(singleItem instanceof GeneralDesignElement)
    {
        GeneralDesignElement gde = (GeneralDesignElement)singleItem;
        System.out.println("Is Frozen:   "+gde.getIs_frozen());
        System.out.println("Cardinality: "+gde.getCardinality());
    }
}
```

On the server side, business objects (**ModelObject** objects) can be added to the **Created**, **Updated** or **Deleted** lists explicitly by the business logic of the given service operation or indirectly through Teamcenter's model event mechanism (in the **ImanTypeImplPublic** and **PSEBulletinBoard** classes). All business objects added to the **ServiceData** object are returned to the calling client with property values as defined by the current object property policy. Deleted objects are returned as just the UID of the deleted business object. Business objects in the **Updated** list are returned with property values beyond what is defined in the current policy. The following conditions determine which properties are returned for **Updated** objects:

- Business object is added to the **ImanTypeImplPublic** update list with an explicit list of property names. Values for these properties are returned in addition to properties from the current policy.
- Business object is added to the **ImanTypeImplPublic** update list with the **TCTYPE_UPDATE_ALL_PROPS** constant. All property values currently loaded in server's memory are returned in addition to properties from the current policy.
- Business object is added to the **PSEBulletinBoard** update list. All property values currently loaded in the server's memory are returned in addition to properties from the current policy.

Service calls made by the rich client also include a list of **Child-Change** objects in the **ServiceData** object. The **Child-Change** list is a subset of the **Updated** list and is determined by the following logic. Any object that is returned from a service operation in the **Updated** list also is returned in the **Child-Change** list if any of the properties returned for the object match the property names defined in the **<Type_Name>_DefaultChildProperties** preference. The previous bullet list describes the property values returned for an updated business object.

The **ServiceData** class also holds partial errors. As most service operations are set based, if an error is encountered while processing a single element in the set, it is reported as a partial error, and processing continues for the remaining elements in the input set. Each partial error has a stack of one or more errors that contributed to the failure. Each error has a code, a localized message, and a severity level. Each partial error may also have an index, a client ID, or a **ModelObject** object attached to the error structure. This is used to identify what input data this partial error is associated with. The index typically is the index to the service operation input array, the client ID is simply a string that is provided by the calling client on one of the input element structures, and the **ModelObject** object is a specific **ModelObject** object the error is associated with. Which is used depends on the context of the service operation, and details are documented in the specific service operation. Partial errors are pulled directly from the **ServiceData** class or from a listener added to the **ModelManager** process:

```
ErrorStack partialError = serviceData.getPartialError ( 0 );
modelManager.addPartialErrorListener( new YourPartialErrorListener() );
```

The **ModelManager** class is accessed from the **Connection** class (**Connection.getModelManager**), is used by the SOA client framework to process **ModelObject** objects returned from service operations, and allows the client application to register different listeners for events related to the **ModelObject** objects. Client applications can instantiate any number of create, change, or delete listeners, each of which is invoked by the **ModelManager** process as data corresponding to each listener type that is returned from the server. The following example shows the use of a **ModelEventListener** component:

```
modelManager.addModelEventListener( new ModelEventListener()
{
    // The client application can implement instances of the
    // localObjectChange method to take appropriate action
    // when object have changed
    public void localObjectChange( ModelObject[] changedObjs )
    {
        for (int i = 0; i < changedObjs.length; i++)
        {
         // The ModelManager will provide a list of ModelObject
         if(singleItem instanceof GeneralDesignElement)
         {
           GeneralDesignElement gde = (GeneralDesignElement)singleItem;
           System.out.println("Is Frozen:   "+gde.getIs_frozen());
           System.out.println("Cardinality: "+gde.getCardinality());
         }
     }
});
```

The **ClientDataModel** class is accessed from the **Connection** class (**Connection.getClientDataModel**), and is used to hold all of the **ModelObject** objects returned by the different service requests. This store of data is cumulative and contains all data returned by the service operations. As different service operations return copies of the same object instance (as identified by UID), the single instance of that object in this store is updated. Objects are only removed from this store if the client application makes explicit calls to remove the data, or if the server has marked the object with a **Delete** event. The model manager also exposes a method to directly retrieve object instances by their UID:

```
ModelObject someItem = clientMetaModel.getObject("xyz123");
if(someItem instanceof GeneralDesignElement)
{
    GeneralDesignElement gde = (GeneralDesignElement)someItem;
    System.out.println("Is Frozen:    "+gde.getIs_frozen());
    System.out.println("Cardinality:  "+gde.getCardinality());
}
```

In general, a client application has little use for accessing **ModelObject** objects in this manner; the application gets the object directly from the service operation response, the **ServiceData** class, or from the **ModelManager** listeners.

The **ClientMetaModel** class is accessed from the **Connection** class (**Connection.getClientMetaModel**) and is used to hold the client-side version of the Teamcenter business model definition (meta model). The meta model information is used by the SOA framework to properly instantiate returned **ModelObject** objects, and is available to the client application to query for information about the meta model. There are two ways to get a business object type definition:

```
Type itemRevType = ClientMetaModel.getType("ItemRevision");
Type someType    = modelObject.getTypeObject();
```

The **Type** class holds metadata about a given type such as its name, localized name, constants, property descriptions, and LOV information. The SOA client framework downloads this data from the server as needed or on demand if it is not in the **ClientMetaModel** class.

Register model object factory

For clients that use strongly typed objects from an extension (non-Foundation) model, you must first register that model's **ModelObjectFactory** object. This requires that the corresponding JAR or DLL files be included with the client. In the code that invokes the extension template's services, the extension factory must first be initialized. This requires code like the following. If this is not done, returned objects are of a more general type, either the nearest parent that is known or the generic **ModelObject** object.

- Java

  ```
  com.teamcenter.soa.client.model.StrongObjectFactory<template-name>.init();
  ```

- C#

```
Teamcenter.Soa.Client.Model.StrongObjectFactorytemplate-name.Init();
```

- C++

```
std::map< std::string, Teamcenter::Soa::Client::ModelObjectFactory*
>* extFactory = template-nameObjectFactory::init();
ModelManager ->registerModelObjectFactory( extFactory );
```

Handling errors with Teamcenter Services

Full service errors

A **ServiceException** exception is thrown only when the implementation of a service operation expects the client application to do special processing when the operation fails as a whole (not only partial errors). For service operations that define that a service exception may be thrown, the client application must surround these service requests with a try/catch block:

```
try
{
    // Execute the service request
    String out = fooService.operationBar ( );
    System.out.println("Service out:"+out);
}
// The service may have thrown an error
catch (ServiceException e)
{
    String[] messages = e.getMessages();
    for (int i = 0; i < messages.length; i++)
    {
       // The Error may have 1 or more sub-errors
       System.out.println( messages[i] );
    }
}
```

Partial errors for Teamcenter model data

The majority of Teamcenter service operations are set-based. The operation may succeed on some of the objects in the set while failing on others. To notify the client application of the failure on the subset of input objects, a **PartialErrors** structure is included in the service data. The client application has several options for accessing partial errors returned from a service operation.

- Partial errors accessed directly from the ServiceData object

  The **ServiceData** object contains a list of partial errors. This list may have 0 or more error stacks on it. Each error stack has a list of one or more errors that contributed to this particular failed action. The service implementation may also set a client ID or associated model object; these are provided to help the client application identify which object this partial error is for.

```
ServiceData manyItems = fooService.operationBar();
```

```
// Loop through one or more of the arrays contained within ServiceData.
// Service documentation should make it clear which arrays may have data
for(int i=0; i<manyItems.sizeOfErrors();i++)
{
    ErrorStack errorStack = manyItems.getPartialError(i);
    String clientId      = errorStack.getClientId();
    ModelObject assocObj = errorStack.getAssociatedObject();
    String[] messages    = errorStack.getMessages();
    for(int j=0; j< messages.length;j++)
    {
       System.out.println( messages[j] );
    }
}
```

- Partial errors accessed from a model manager listener

  The model manager provides the **PartialErrorListener** object to notify the client application when partial errors have been returned from any Teamcenter service operation. The client application may create any number of implementations of this listener. The model manager invokes each instance of the listeners as appropriate data is returned from a service operation.

```
modelManager.addPartailErrorListener( new PartialErrorListener()
{
    public void handlePartialError(ErrorStack[] partialErrors )
    {
        // Loop through the list of Partial Errors.
        for(int i=0; i<partialErrors.length;i++)
        {
            ErrorStack errorStack = partialErrors[i];

            // The Partial Error may have a client ID or a Model
            // Object associated with it
            System.out.println("Error for client Id "+
                               errorStack.getClientId());
            Modelbject assocObj = errorStack.getAssociatedObject();

            // Each Partial Error may have 1 or more sub-errors
            String[] messages    = errorStack.getMessages();
            for(int j=0; j< messages.length;j++)
            {
                 System.out.println( messages[j] );
            }
        }
    }
});
```

- Partial errors for non-Teamcenter model data

  For set-based service operations that do not return a **ServiceData** object, partial errors may be returned as part of any returned data structure.

```
BarResponse out = fooService.operationBar();
for(int i=0; i<out.partialErrors.size();i++)
{
    ErrorStack errorStack = out.partialErrors.getErrorStack(i);
    String clientId      = errorStack.getClientId();
```

```
        ModelObject assocObj = errorStack.getAssociatedObject();
        String[] messages    = errorStack.getMessages();
        for(int j=0; j< messages.length;j++)
        {
           System.out.println( messages[j] );
        }
    }
```

Object property policy

The object property policy defines which properties are returned from a service operation for a given object type. The business logic of a service operation determines what business objects are returned, while the object property policy determines which properties are returned on each business object instance. This allows the client application to determine how much or how little data is returned based on how the client application uses those returned business objects. The policy is applied uniformly to all service operations. The client application manages what policy is to be used at any given time during the session.

By default, all applications use the default object property policy. It is this policy that is applied to all service operation responses until the client application changes the policy. Siemens Digital Industries Software strongly recommends that all applications change the policy to one applicable to the client early in the session. There are two service operations available to change the policy:

```
SessionService.setObjectPropertyPolicy( String policyName );
SessionService.setObjectPropertyPolicy( ObjectPropertyPolicy policy );
```

policyName	Specifies the name of the policy file without the **.xml** extension. The file must exist on the Teamcenter server volume at *TC_DATA***/soa/policies/***policy-name***.xml**.
policy	Specifies an instance of the **ObjectPropertyPolicy** class. This policy can be read from a persisted file on the client host or created programmatically.

The client application can use either of these operations or a combination of both to set the policy throughout the session. Using the **ObjectPropertyPolicy** class versus the named policy on the server gives the client application greater control of defining a policy. The **ObjectPropertyPolicy** class can be constructed in the client based on user preferences or user interaction and persisted to the local file system, enabling the ability to define a policy for a particular user versus forcing all users of the application to have the same policy.

The **ObjectPropertyPolicyManager** class has methods to help the client application manage the policy. This class is accessed using the **Connection.getObjectPropertyPolicyManager** method.

```
String addPolicy( String policyName, PolicyStyle style );
void setPolicy ( String policyName );
void setPolicyPerThread ( String policyName );
ObjectPropertyPolicy getPolicy( String policyName );
```

The set methods are convenience methods to the **SessionService** operation and may result in a server trip if the named policy has not previously been set with a call to the **SessionService** operation. More information about these methods can be found in the *Services Reference*.

> Note:
>
> The *Services Reference* is available on Support Center.

The **ObjectPropertyPolicy** class is used to define an object property policy. The object property policy is a list of Teamcenter classes and types and properties associated with those classes and types. The properties defined in a parent class are inherited by child classes. There are number of methods on the **ObjectPropertyPolicy** class to add and remove types and properties. Following is an example of creating a policy for the **WorkspaceObject**, **Folder** and **BOMLine** types:

```
ObjectPropertyPolicy policy = new ObjectPropertyPolicy();
policy.addType( "WorkspaceObject", new String[]{"object_string",
"object_type"});
policy.addType( new PolicyType("Folder",
                               new String[]{"contents"},
                               new String[]
{PolicyProperty.WITH_PROPERTIES}));
policy.addType( "BOMLine", new String[]{"bl_has_children",
"bl_is_variant"});
```

Types inherit properties defined on parent types, so the **Folder** type includes the **object_string** and **object_type** properties defined on the **WorkspaceObject** type. The object property policy does not make any distinction between persisted types (**POM_Object**) and run-time types like **BOMLine**. Modifiers may be added to a property, type, or the entire policy. These modifiers give extra instructions on how property data is returned for the given type/property. The **WITH_PROPERTIES** modifier on the **Folder** contents property will also return the property values from the business objects referenced by the **Folder** contents property. By default, referenced business objects are returned without any property values. There are five valid modifiers:

- **PolicyProperty.WITH_PROPERTIES**

  By default, the object property policy is applied only to the objects explicitly returned by the service implementation. If the service returns a **User** object and the **home_folder** property is part of the policy, the referenced **home_folder** object is added to the return data, but without any associated properties. By setting this modifier to true on the **home_folder** property, properties for the home folder are also returned.

- **PolicyProperty.EXCLUDE_UI_VALUE**

  By default, the object property policy returns both the database and user interface value for each property. By setting this modifier to **true**, only the database values of a property are returned and the user interface value is excluded from the return data.

- **PolicyProperty.INCLUDE_MODIFIABLE**

By default, the object property policy does not return the **Is Modifiable** flag with each property value. The **Is Modifiable** flag is the instance based flag versus the **Meta Model** flag defined on the **PropertyDesciption** object. By setting this modifier to **true**, the **Is Modifiable** flag is returned.

- **PolicyProperty.UI_VALUE_ONLY**

  By default, the full property information is returned for the named property. This includes the database value and the display or user interface value. With this flag set to **true**, only the user interface value is returned. This flag takes precedence over any other flags set in the policy for the named property.

- **PolicyProperty.EXCLUDE_PARENT_PROPERTIES**

  By default, the properties defined on the parent type are included in the current type. With this property set to **true**, properties defined on the parent types are excluded.

The object property policy may be persisted to the file system on the client host machine with the following methods:

```
ObjectPropertyPolicy.save( String filePath );
ObjectPropertyPolicy.load( String filePath );
```

The policy is persisted as an XML file using the following XML schema:

```
<xs:schema
    xmlns:xs=http://www.w3.org/2001/XMLSchema
    xmlns:tns="http://teamcenter.com/Schemas/Soa/ObjectPropertyPolicy"
    targetNamespace="http://teamcenter.com/Schemas/Soa/ObjectPropertyPolicy"
    elementFormDefault="qualified" attributeFormDefault="unqualified">

    <xs:element name="ObjectPropertyPolicy">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="ObjectType" type="tns:ObjectType"
maxOccurs="unbounded"/>
            </xs:sequence>
            <xs:attribute name="withProperties"          type="xs:string"
use="optional" />
            <xs:attribute name="excludeUiValues"         type="xs:string"
use="optional" />
            <xs:attribute name="includeIsModifiable"     type="xs:string"
use="optional" />
            <xs:attribute name="uIValueOnly"             type="xs:string"
use="optional" />
            <xs:attribute name="excludeParentProperties" type="xs:string"
use="optional" />
        </xs:complexType>
    </xs:element>

    <xs:element name="ObjectType">
        <xs:complexType>
            <xs:sequence>
                <xs:element name="Property" type="tns:Property" maxOccurs="unbounded"/>
```

```
            </xs:sequence>
            <xs:attribute name="name"                    type="xs:string"
use="required" />
            <xs:attribute name="withProperties"          type="xs:string"
use="optional" />
            <xs:attribute name="excludeUiValues"         type="xs:string"
use="optional" />
            <xs:attribute name="includeIsModifiable"     type="xs:string"
use="optional" />
            <xs:attribute name="uIValueOnly"             type="xs:string"
use="optional" />
            <xs:attribute name="excludeParentProperties" type="xs:string"
use="optional" />
        </xs:complexType>
    </xs:element>

    <xs:element name="Property">
        <xs:complexType>
            <xs:attribute name="name"                    type="xs:string"
use="required" />
            <xs:attribute name="withProperties"          type="xs:string"
use="optional" />
            <xs:attribute name="excludeUiValues"         type="xs:string"
use="optional" />
            <xs:attribute name="includeIsModifiable"     type="xs:string"
use="optional" />
            <xs:attribute name="uIValueOnly"             type="xs:string"
use="optional" />
            <xs:attribute name="excludeParentProperties" type="xs:string"
use="optional" />
        </xs:complexType>
    </xs:element>
</xs:schema>
```

More information about the **ObjectPropertyPolicy** class and related classes can be found in the *Services Reference* developer reference, available on Support Center.

For policies defined on the Teamcenter server side, there are no APIs to read or manipulate the policy. The source XML file (*TC_DATA***/soa/policies/***policy-name***.xml**) is created manually using the text or XML editor of your choice. The server-side policy XML files conform to the preceding XML schema with two additions: a policy can include other policy files, and types may use **Preference** variables to define a set of property names:

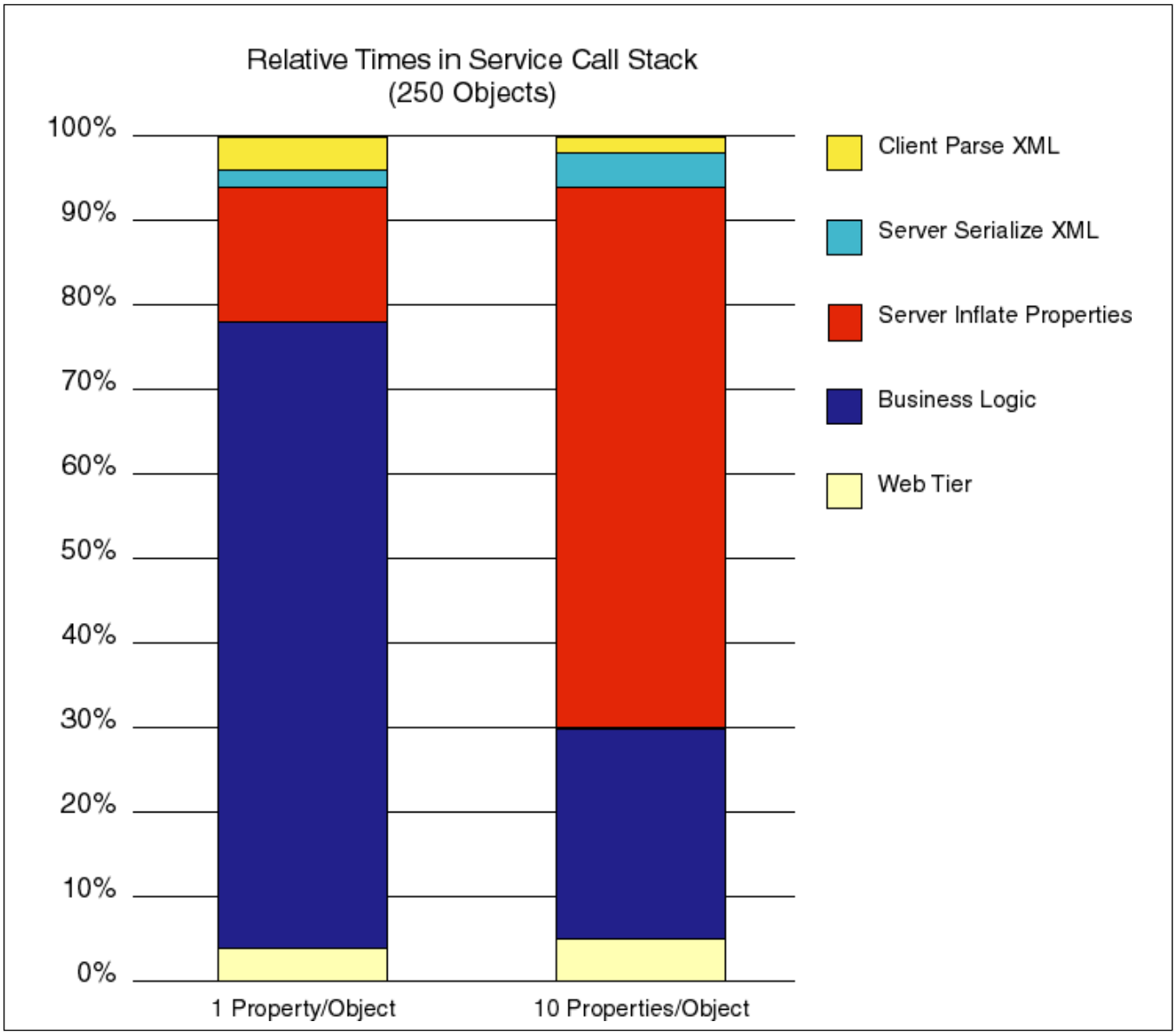
```
<ObjectPropertyPolicy>

     <Include file="site/MySitePolicy.xml ">

    <ObjectType name="WorkspaceObject">
        <Property name="object_string" />
        <Property name="object_type"   />
    </ObjectType>

    <ObjectType name="Folder">
        <Property name="contents" withProperties="true" />
    </ObjectType>
    <ObjectType name="BOMLine">
            <Property   name="bl_has_children" />
```

```
            <Property   name="bl_is_variant"   />
            <Preference name="PortalPSEColumnsShownPref" />
        </ObjectType>
    </ObjectPropertyPolicy>
```

The **Include** element includes types and properties defined in the site **MySitePolicy.xml** file (*TC_DATA***/soa/policies/site/MySitePolicy.xml**). The **Preference** element of the **BOMLine** type uses the list of property names defined in the **PortalPSEColumnsShownPref** preference.

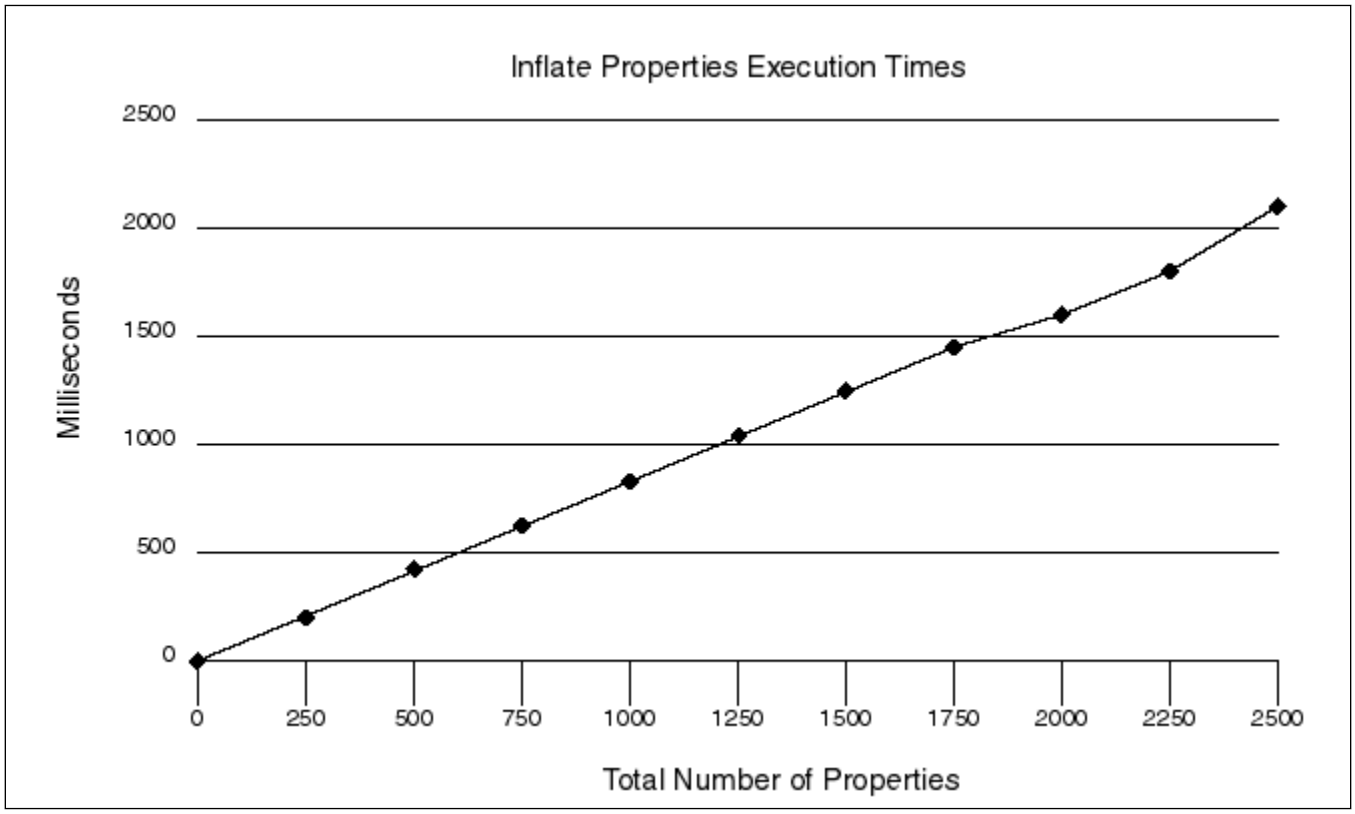
The number of property values included in the response of a service operation is one of the biggest impacts on overall performance of the services. For that reason it is important for the client application developer to have a clear understanding of the object property policy.

The following figure compares two service requests each returning 250 objects. The first service request uses an object property policy that only defines one property for the given object type, for a total of 250 property values being returned, while the second service requests uses an object property policy that defines 10 properties for the given object type, for a total of 2,500 property values being returned. The red section of the graph shows the relative time spent retrieving property values. This is where the Teamcenter Services framework uses the current policy to retrieve the desired properties for each object in the **ServiceData** object.



Object property policy impact on performance

It does not matter how the properties are distributed across the returned objects, whether each object has 10 properties, or that some object types have 3 properties and others have 15. All that matters is the total number of properties that the property policy manager of the Teamcenter Services framework must retrieve from the database and add to the service response. The following figure shows a very linear increase in execution time with the increase in total number of properties returned.



Object project policy effect on execution time

Applications should use policies that return the data needed for the use cases they support and should not retrieve too many properties in case they are required for a subsequent operation.

Improving services performance

You can improve the performance of your Teamcenter Services applications by applying the following adjustments:

- Set start and stop refresh boundaries.

  To prevent Teamcenter Services from refreshing each object as it is written to the **ServiceData** response, an **in_operation** bracket is established for each individual service request. Objects are refreshed only once inside the bracket, suppressing redundant refresh operations. To begin a logical bracket, the client may issue a **startOperation** request. The bracket remains in effect until the client issues a **stopOperation** request with the same operation ID as returned by the **startOperation** request. For example:

  ```
  std::string Teamcenter::Soa::Core::_2009_03::Session::startOperation(operation-id);
  Teamcenter::Soa::Core::_2009_03::Session::stopOperation( std::string id operation-id);
  ```

  To ensure that clients do not inadvertently leave an active bracket open indefinitely, each bracket is given an expiration time stamp. The **SOA_MaxOperationBracketInactiveTime** preference sets the

duration of the inactivity time-out period for objects in a bracket, with a default of 60 seconds. The **SOA_MaxOperationBracketTime** preference sets the duration of the expiration period, with a default of 600 seconds.

- Set the object property policy **asAttribute** flag.

  To improve performance, you can set an **asAttribute** flag to **true** in a property policy if the property is an **ImanPropertyAttribute** property type. This obtains values for a property element using the lower level **getValue** operation, which does not modify the attribute value, thereby improving performance.

  Use the **asAttribute** flag only as a performance optimization, when the application and site clearly intend that no extra processing for the property is necessary. To aid debugging, during parsing of the property policy file, the initialization journals all occurrences of the **asAttribute** XML attribute.

- Call the C++ memory manager.

  Clients can register their memory manager. An API is exposed to register callback functions with the framework to manage memory. Instrumentation can be added to track memory allocated for client data model (CDM) objects.

  Before creating a connection object, the client application must implement an allocator class derived from the **Teamcenter::Soa::Common::MemoryAllocator** class and call the **MemoryManager::initializeMemoryManager** factory method.

  When a connection object is created, the Teamcenter Services framework checks to see if a memory manager is initialized by the client application; if not, it initializes with a default allocator.

Teamcenter services and TCCS

How to integrate Teamcenter Services with TCCS

Teamcenter client communication system (TCCS) provides secure communications through a firewall using a proxy to a Teamcenter server.

> Note:
>
> To install TCCS on a Teamcenter server, run the *installation-source***additional_applications\tccs_install\tccs.exe** file.

When you must connect an SOA client to a Teamcenter server running TCCS, use the **SoaConstants.TCCS** protocol argument to instantiate a connection object.

The TCCS protocol uses the following connection options:

- **TCCS_ENV_NAME**

Specifies the environment name to which the SOA client wants to connect.

- **TCCS_USE_CALLBACK**

  Specifies if the proxy authentication challenge must be handled by the client as opposed to the TCCS process.

- **TCCS_HOST_URL**

  Contains an arbitrary URL specified by the client. TCCS uses this URL instead of the one specified in the TCCS configuration.

The TCCS protocol uses the following methods:

- **setProxyCredentialProvider (CredentialProvider** *provider***)**

  Sets the client application's credential provider to handle proxy authentication challenges in the client. Replace *provider* with the credential provider name. If this method is not set, a default credential provider in the client is used. (A default credential provider is available only in Java.)

- **getEnvsForVersion (string** *expression***)**

  Gets a list of TCCS environments matching the version expression. Replace *expression* with the version defined in the TCCS environment. Multiple versions can be specified in the expression using the **OR** separator, for example, **8.3|9.0**. The ***** wildcard can be specified at the end of the input string.

- **getEnvironment (string** *environment-name***)**

  Gets environment data based on the TCCS environment name. Replace *environment-name* with the corresponding TCCS environment name.

- **getEnvironmentsForURL (string** *URL***)**

  Gets environment data for the specified server URL. Replace *URL* with the name of the server URL specified in the TCCS environment.

The TCCS protocol provides the following APIs:

- Environment name APIs

  Define the environment to connect to. To integrate with TCCS, the client application must provide the TCCS environment name when initializing the connection object. The SOA connection class available in the SOA framework library provides the following APIs to query a list of all environments defined in the TCCS configuration:

  - Java

```
Connection.java : public static ArrayList < Environment > getEnvironments()
throws TSPException
```

- C#

```
Connection.cs : public static ArrayList getEnvironments()
```

- C++

```
Connection.cxx : std::vector < const Teamcenter::Net::TcServerProxy::
Admin::Environment* >
   Teamcenter::Soa::Client::Connection::getEnvironments()
```

- Query environments APIs

  Define a query for environments to connect to. An API to query list environments based on filter expression is also available in the SOA connection class. The input expression corresponds to the version defined in TCCS environment. The following APIs can be used to query for the environments matching the input expression:

  - Java

```
Connection.java : public static ArrayList < Environment >
getEnvsForVersion(final String expression)
   throws TSPException
```

  - C#

```
Connection.cs : public static ArrayList getEnvsForVersion(string expression)
```

  - C++

```
Connection.cxx : std::vector < const Teamcenter::Net::TcServerProxy::
Admin::Environment* >
   Teamcenter::Soa::Client::Connection::getEnvsForVersion(const std::string
expression)
```

An SOA-based client application can get the environment name for each of the queried environments in the list using the **getName()** method of the environment object. The client application can either display the environment names to the user and ask the user to select the environment name to be used with a new session, or use some other setting or preference for determining the environment that needs to be used. The application must supply the environment name to be used with TCCS when initializing the SOA connection. To use TCCS for requests in a four-tier deployment, you must specify the protocol as **TCCS** in the SOA connection constructor:

```
public Connection( String hostPath, HttpState cookieManager,
CredentialManager credentialManager,
   String binding, String protocol, boolean useCompression)
```

hostPath can be **null**. and its value is ignored because TCCS uses the host path provided in the TCCS environment definition.

cookieManager can be **null**, and its value is ignored because TCCS maintains its own pool of HTTP state objects.

protocol specifies the value of TCCS to be used for integrating with TCCS.

Set options as follows:

```
setOption( String optionName, String value)
```

- **TCCS_ENV_NAME**

  Provide the environment name to be used with the SOA connection.

  Replace *optionName* with **TCCS_ENV_NAME**, and set *value* to the TCCS environment name to be used with this connection.

- **TCCS_USE_CALLBACK**

  Specify if the proxy authentication challenge must be handled by the client as opposed to TCCS process.

  Replace *optionName* with **TCCS_USE_CALLBACK** and set *value* to **true** or **false**.

- **TCCS_HOST_URL**

  Specify an arbitrary URL. Client applications intending TCCS to use this arbitrary server URL must specify this option. TCCS uses this URL instead of the one specified in the TCCS configuration.

  Replace *optionName* with **TCCS_HOST_URL** and set *value* to **true** or **false**.

> Note:
>
> The existing constructor that takes the SSO login service URL and the SSO AppID also supports applet free usage if the SSO login service URL is specified with the appropriate format. You get applet free SSO when the SSO login service URL is provided with **/tccs** at the end of the URL.

Sample code to integrate a client with TCCS

There are two ways to specify the TCCS environment name when you create the **Connection** object configured for TCCS:

Using the hostPath argument

```
CredentialManager credentialMgr = new YourAppCredentialManager();
ExceptionHandler  expHandler    = new YourAppExceptionHandler()

Connection connection = new Connection("tccs://teamcenter-4tier",
                        credentialMgr, SoaConstants.REST,
SoaConstants.TCCS);
ExceptionHandler  expHandler = new YourAppExceptionHandler();

connection.setExceptionHandler( expHandler );
```

Using the TCCS_ENV_NAME option

```
CredentialManager credentialMgr = new YourAppCredentialManager();
ExceptionHandler  expHandler    = new YourAppExceptionHandler();

Connection connection = new Connection("", credentialMgr,
                        SoaConstants.REST, SoaConstants.TCCS);

connection.setOption( Connection.TCCS_ENV_NAME, "teamcenter-4tier");
connection.setExceptionHandler( expHandler );
```

Use a proxy credential provider in a client application

Clients integrating with TCCS can provide their own proxy credential provider to handle proxy authentication challenges. This custom credential provider is used instead of the one provided in TCCS.

The connection class provides APIs to support the use of a proxy credential provider. Perform the following to use your own proxy:

1. Set the **TCCS_USE_CALLBACK** connection to **true**:

   ```
   connection.setOption(Connection.TCCS_USE_CALLBACK, "true");
   ```

   This option allows the client application to use its own credential provider and not the one provided in TCCS.

2. Set the custom credential provider:

   ```
   connection.setProxyCredentialProvider(new
   MyProxyCredentialProvider());
   ```

   The **MyProxyCredentialProvider** in the sample is the custom credential provider that must implement the **com.teamcenter.net.tcserverproxy.client.CredentialProvider** interface in TCCS.

3. Ensure the **getCredential** method in the implemented class returns the **Credential** object with forward proxy user ID and password.

   The following example shows the method with hardcoded credentials:

```
class MyProxyCredentialProvider implements CredentialProvider
{
    @Override
    public Credential getCredential(final AuthScope authscope)
throws CredentialsUnavailableException
    {
      String userID = "user-name";
      String passwd = "password";
               Credential proxyCreds = new Credential(userID,
passwd);
      return proxyCreds;
   }
 }
```

   This class could return the credentials by other means like reading the credentials from user input or through a logon dialog box.

Handling time and date settings

Within Teamcenter, dates and times are represented with different structures and classes. However, as the date and time values are moved across Teamcenter, the absolute value is maintained across programming languages, time zones, and daylight savings time. The following table lists the different classes used to represent dates and times in Teamcenter.

Teamcenter component	Class	Header file	Library
POM/database	**date_t (struct)**	**unidefs.h**	Teamcenter library
SOA service implementation and ITK	**Teamcenter::DateTime**	**DateTime.hxx**	**libbase_utils.dll**
SOA transport layer	**xsd:date**	**"yyyy-MM-dd'T'HH:mm:ssZ"**	NA
Java client	**java.util.Calendar**	NA	JRE
Rich client	**java.util.Date** or **java.util.Calendar**	NA	JRE
C++ client	**Teamcenter::Soa::**	**teamcenter/soa/common/**	**libtcsoacommon**

Teamcenter component	Class	Header file	Library
	Common::DateTime	**DateTime.hxx**	
C# client	**System.DateTime**	NA	.NET framework

The POM, ITK, and C++ client represent the date in the time zone of the host, while the Java and C# client have time zone information included in the representation of the date and time. At the SOA transport layer, the client and server may be in different physical locations that are in different time zones. As date and time values move from the server to the client through the SOA transport layer, the date and time is serialized using the formatting defined by an XML schema for dates. This includes time zone information in the serialized date and time value, preserving the absolute value of the date and time value as it moves from server to client.

The POM layer has the concept of a null date. The null date is a constant that has a representation in each of the Teamcenter components.

Teamcenter component	Null date
POM/database	**date_t POM_null_date();**
SOA service implementation and ITK	**Teamcenter::DateTime Teamcenter::DateTime::getNullDate();**
SOA transport layer	**0001-01-01T00:00:00+00:00**
Java client	**null**
C++ client	**Teamcenter::Soa::Common::DateTime Teamcenter::Soa::Common::DateTime::getNullDate();**
C# client	**System.DateTime Teamcenter.Soa.Client.Model.Property.NullDate;**

While the internal representation of the null date may differ from class to class, the null date itself is preserved as it is passed from component to component. A null value in a Java client can be passed to the server using the SOA transport layer (an argument on a service request), while in the Teamcenter server POM layer it is represented by a **date_t** instance that is equivalent to the constant returned from the **POM_null_date()** function.

Character encoding in C++

C++ clients can specify the type of encoding through the **OPT_ENCODING_TYPE** connection option. For example:

```
Teamcenter::Soa::Client::Connection::setOption(OPT_ENCODING_TYPE,encoding-type)
```

If this option is not set, the conversion from the local code page to UTF-8 (while encoding) and UTF-8 to local code page (while decoding) takes place.

The **getTcSessionInfo** service gets the server encoding information that is added to the **TcSessionInfo** response. This response object has an **extraInfo** array that contains the server encoding value. The encoding value can be retrieved at the client application using the following:

```
vector<ExtraInfo> > extraInfoArray = infoResponse->getExtraInfoArray();
map<string,string> extraInfo;
for( size_t i=0; i<extraInfoArray.size(); ++i )
{
ExtraInfo element = extraInfoArray[i];
extraInfo[ element->getKey()] = element->getValue();
}

ExtraInfo::iterator it = extraInfo.find( "TcServerEncoding" );
if( it != extraInfo.end() )
        {
String serverEncoding = it->second;
}
```

3. Teamcenter Services organization

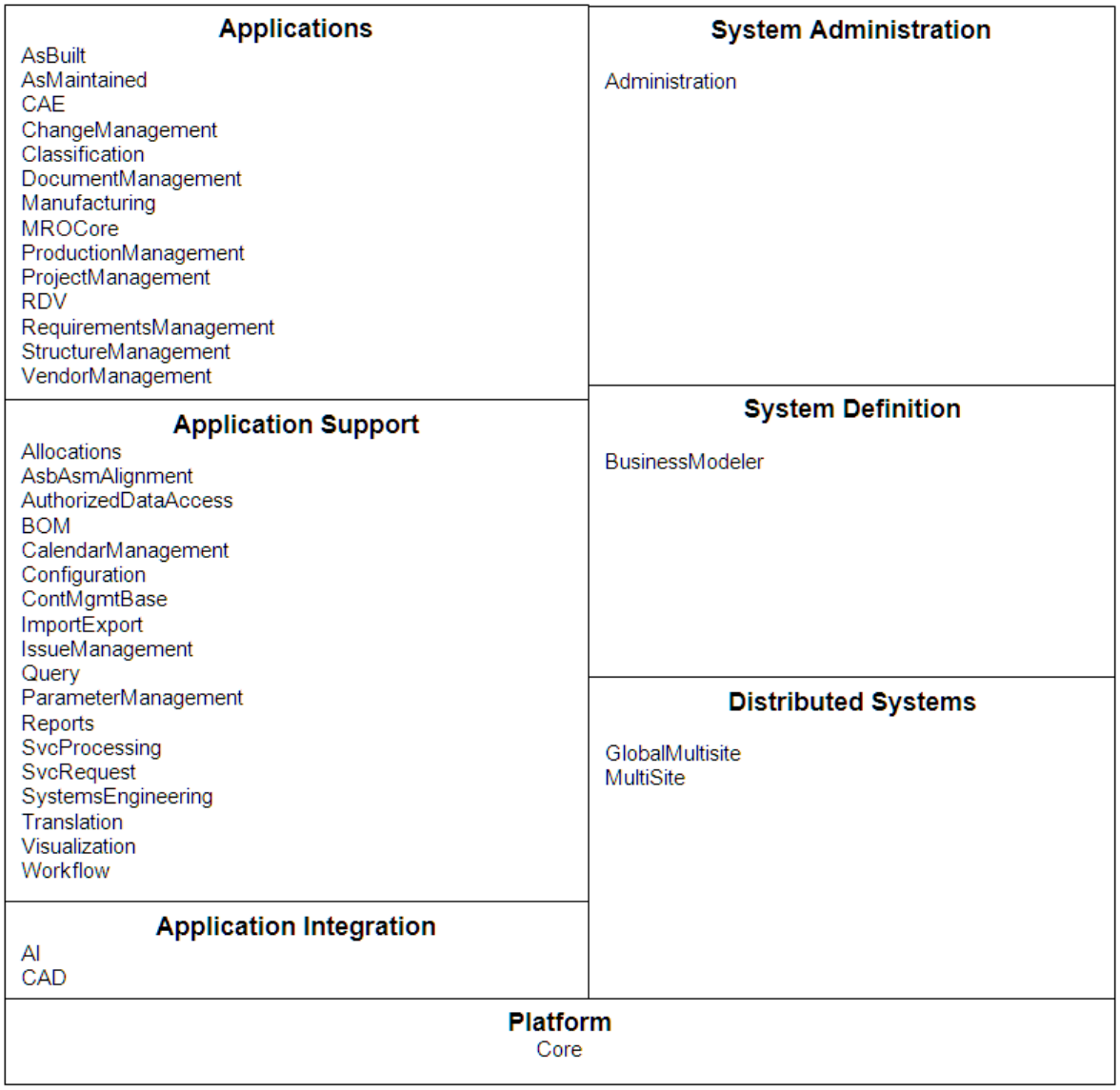
Services functional groups

The set of Teamcenter Services APIs is vast and will continue to grow and expand with each subsequent version. Navigating the list of available capabilities can be a daunting task. To address this, Teamcenter Services APIs are organized as a hierarchy of functional groups, libraries, services, and operations:

Functional groups
  Libraries
    Services
      Operations

At the highest level are the *functional groups*, a small number of coarse-grained categories into which the various API code libraries are organized. The functional groups merely allow users to quickly determine where to look for the feature that they need at a very high level. Functional groups only serve to provide clarity and grouping within the documentation, and they are not formally used within the building of any code. A functional group can have any number of libraries within it.

The following figure illustrates the service library organization in the six functional groups.



Teamcenter service functional organization

The functional groups are as follows:

- Applications

  Contains libraries for specific capabilities and industry solutions.

- Application Support

  Contains libraries that serve the common needs of day-to-day end-user applications. Available services and operations support standard mechanisms for system queries and searches, report definition and generation, and creation and maintenance of calendars used in project management and workflow applications.

- Application Integration

  Contains libraries for connecting to outside applications. Teamcenter applications are generally deployed as one part of a corporate information environment that includes a variety of non-PLM systems and applications. Applications and information managed by Teamcenter must be easily accessible and interoperate smoothly with other systems and applications.

- System Administration

  Contains libraries related to the day-to-day administration of a Teamcenter system or deployment. Along with Teamcenter Platform services and Teamcenter Application Support services, the operations exposed can be used in a variety of custom administration scenarios that may be important to your organization.

- System Definition

  Contains libraries used by the Teamcenter Business Modeler IDE and other tools to define and customize a particular Teamcenter installation or system deployment.

- Distributed Systems

  Contains libraries for Global Services and those portions of Global Services that address connectivity and functionality across and between Teamcenter installations. Other portions of Global Services address integration with non-PLM systems and are more appropriately included in the Application Integration functional area.

- Platform

  Contains libraries that are fundamental to how data is stored, managed, and accessed in Teamcenter. These libraries are independent of any particular Teamcenter application space. They make up the core of functionality that all other application areas are built upon.

Services libraries

A *library* corresponds to a compiled code artifact such as a shared library (DLL or SO) or archive (JAR), and contains *services* relating to particular Teamcenter functionality or capabilities. Within each service are a set of *operations* that make up the actual callable methods on the API. To fully qualify an operation, the notation *library::service::operation* can be used.

The Teamcenter Services client libraries are located in the **soa_client.zip** file on the Teamcenter software distribution image. To get started using services, after you have extracted the ZIP file, go to **soa_client/Help.html**. The API for services and operations are documented in the *Services Reference*.

> Note:
>
> The *Services Reference* is available on Support Center.

After you extract the **soa_client.zip** file, you can locate the extracted library files in the Java (**java\libs**), C++ (**cpp\libs**), and C# (**net\libs**) library subdirectories.

Following are the files used for each library:

- Java

  TcSoa*library-name***Strong.jar**
  TcSoa*library-name***Loose.jar**

- C++

  libtcsoa*library-name***strongmngd.dll**

- C#

  TcSoa*library-name***Strong.dll**

For example, the following files are used for the **Core** library:

- Java

  TcSoaCoreStrong.jar
  TcSoaCoreLoose.jar

- C++

  libtcsoacorestrongmngd.dll

- C#

  TcSoaCoreStrong.dll

Each service in a library follows this namespace standard:

- Java

  `com.teamcenter.services.strong.`*library-name*`.`*service-name*`Service`

- C++

`Teamcenter::Services::`*library-name*`::`*service-name*`Service`

- C#

  `Teamcenter.Services.Strong.`*library-name*`.`*service-name*

For example, following are the namespaces for the **LOV** service in the **Core** library:

- Java

  `com.teamcenter.services.strong.core.LovService`

- C++

  `Teamcenter::Services::Core::LovService`

- C#

  `Teamcenter.Services.Strong.Core.Lov`

A. Teamcenter services preferences

SOA_MaxOperationBracketInactiveTime

DESCRIPTION

Specifies the maximum time in seconds for an operation bracket without active server requests.

For more information, see *Teamcenter Services*.

VALID VALUES

Single positive integer.

DEFAULT VALUES

60

DEFAULT PROTECTION SCOPE

Site preference.

NOTES

If an SOA **startOperation** request is issued, the operation bracket should be terminated with a **stopOperation** request. If the client leaves a bracket open without any server requests for this period of time, the bracket is automatically terminated upon the next SOA request.

SOA_MaxOperationBracketTime

DESCRIPTION

Specifies the maximum time in seconds for an operation bracket to suppress redundant refreshes.

For more information, see *Teamcenter Services*.

VALID VALUES

Single positive integer.

DEFAULT VALUES

600

DEFAULT PROTECTION SCOPE

Site preference.

NOTES

If an SOA **startOperation** request is issued, the operation bracket should be terminated with a **stopOperation** request. If bracket lasts longer than specified by this preference, the bracket is automatically terminated.