BEA WebLogic Server

Using WebLogic File Services

WebLogic Server 6.0
Document Edition 1.0
December 2000

Copyright

Restricted Rights Legend

Trademarks or Service Marks

Using WebLogic File Services

Document Edition	Date	Software Version
6.0	December 2000	BEA WebLogic Server 6.0

Contents

Preface

This document explains how to use WebLogic File services for client-side access to native operating system files on the server.

This document covers the following topics:

- Chapter 1, “Using WebLogic File Services,” introduces WebLogic File, describes the WebLogic File API, and provides instructions on programming with WebLogic File.

What You Need to Know

This document is intended primarily for application developers who are interested in reading and writing files within Java applications that run in the WebLogic Server environment. It assumes a familiarity with the WebLogic Server platform, Java and Java 2 enterprise Edition (J2EE) programming, and file I/O concepts.

e-docs Web Site

The BEA WebLogic Server product documentation is available on the BEA Systems, Inc. corporate Web site. From the BEA Home page, click the Product Documentation button or go directly to the “e-docs” Product Documentation page at http://e-docs.bea.com.

How to Print the Document

You can print a copy of this document from a Web browser, one file at a time, by using the File—>Print option on your Web browser.

A PDF version of this document is available on the WebLogic Server documentation Home page on the e-docs Web site (and also on the documentation CD). You can open the PDF in Adobe Acrobat Reader and print the entire document (or a portion of it) in book format. To access the PDFs, open the WebLogic Server documentation Home page, click the PDF Files button, and select the document you want to print.

If you do not have the Adobe Acrobat Reader installed, you can download it for free from the Adobe Web site at http://www.adobe.com/.

Contact Us!

Your feedback on the BEA WebLogic Server documentation is important to us. Send us e-mail at **docsupport@bea.com** if you have questions or comments. Your comments will be reviewed directly by the BEA professionals who create and update the WebLogic Server documentation.

In your e-mail message, please indicate that you are using the documentation for the BEA WebLogic Server and include the release number.

If you have any questions about this version of BEA WebLogic Server, or if you have problems installing and running BEA WebLogic Server, contact BEA Customer Support through BEA WebSUPPORT at www.bea.com. You can also contact Customer Support by using the contact information provided on the Customer Support Card, which is included in the product package.

When contacting Customer Support, be prepared to provide the following information:

- Your name, e-mail address, phone number, and fax number
- Your company name and company address
- Your machine type and authorization codes

- The name and version of the product you are using
- A description of the problem and the content of pertinent error messages

Documentation Conventions

The following documentation conventions are used throughout this document.

Convention	Item
boldface text	Indicates terms defined in the glossary.
Ctrl+Tab	Indicates that you must press two or more keys simultaneously.
italics	Indicates emphasis or book titles.

`monospace text`	Indicates code samples, commands and their options, data structures and their members, data types, directories, and filenames and their extensions. Monospace text also indicates text that you must enter from the keyboard.
	Examples:
	`import java.io.Serializable;`
	`public String getName();`
	`\tux\data\ap`
	`.doc`
	`tux.doc`
	`BITMAP`
	`float`

`monospace boldface text`	Identifies significant words in code. *Example*: `void` **`commit`** `( )`
`monospace italic text`	Identifies variables in code. *Example*: `String` *`expr`*

Convention	Item
UPPERCASE TEXT	Indicates device names, environment variables, and logical operators. *Example*s: LPT1 SIGNON OR
`{ }`	Indicates a set of choices in a syntax line. The braces themselves should never be typed.
`[ ]`	Indicates optional items in a syntax line. The brackets themselves should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
`|`	Separates mutually exclusive choices in a syntax line. The symbol itself should never be typed.
`...`	Indicates one of the following in a command line: ■ That an argument can be repeated several times in a command line ■ That the statement omits additional optional arguments ■ That you can enter additional parameters, values, or other information The ellipsis itself should never be typed. *Example*: `buildobjclient [-v] [-o name ] [-f file-list]... [-l file-list]...`
. . .	Indicates the omission of items from a code example or from a syntax line. The vertical ellipsis itself should never be typed.

1 Using WebLogic File Services

This section describes the WebLogic File services and includes the following topics:

- Overview of WebLogic File Services
- WebLogic File API Reference
- Programming with WebLogic File Services

Overview of WebLogic File Services

WebLogic File provides high-speed, client-side access to native operating system files on the server. The client API extends the lowest-common-denominator capabilities of Java (`java.io.InputStream` and `java.io.OutputStream`), which allows it to be used seamlessly in existing code, with additional services that are specific to manipulating remote files.

As a service, WebLogic File also has access to all of the other WebLogic facilities, like logging, instrumentation, and workspaces. All WebLogic component-based services, including File services, are integrated into the WebLogic framework and can share access and resources. Their APIs share many common aspects that make building a complex networked application easier. Your application may use a variety of these services, all of which can share access to objects and client resources.

With WebLogic File, as with other WebLogic services, the client uses factory methods to generate `T3FileInputStream` and `T3FileOutputStream` objects. These classes extend the standard Java `InputStream` and `OutputStream` classes, allowing them to be plugged into existing client code. They also provide additional methods that are specific to remote file streams.

WebLogic File enhances read and write performance over a network by transmitting data in buffers whose size is independent of the size of the requests, and by using `readAhead` and `writeBehind` buffering. The implementation increases the rate of data transfer in several ways.

- Data is transmitted in buffers whose size is independent of the size of application requests. An application can make many small requests without adversely affecting performance.
- The client does read ahead; that is, it automatically requests buffers ahead of the application. While an application is processing a buffer of data, the next buffer is being simultaneously retrieved.
- The client does write behind; that is, it allows the application to write buffers beyond what has been flushed to the disk on the server. While an application is preparing a buffer of data, the previous buffers are being simultaneously written to the disk. A flush operation blocks on the client until an acknowledgment has been received that all outstanding buffers have been flushed.

An application may specify the transfer buffer size, the number of buffers of read ahead, and the number of buffers of write behind, or it may rely on default values. The default buffer size is 100K, and the default number of buffers for both read ahead and write behind is 1.

The defaults set by WebLogic File are usually the best choice for maximum speed. If you decide not to use the defaults, here are some hints for choosing other values.

- *Setting the buffer size*. In general, the larger the transfer buffer size, the greater the raw speed of the transfer. The difference can be significant; using a 1K buffer might be almost an order of magnitude slower than a 100K buffer. However, larger buffers require more memory on the client side, so you need to determine the most effective settings for your configuration.
- *Setting readAhead and writeBehind buffers*. The best value for `readAhead` and `writeBehind` depends on the rate at which your application processes buffers relative to the transfer speed. With a consistently slower application, a single buffer of `readAhead` and `writeBehind` provides the maximum benefit. A

consistently faster application does not benefit at all from increasing `readAhead` and `writeBehind`. Thus, the default value of 1 works well in most cases. However, if your application varies the rate at which it processes buffers, you may want to increase `readAhead` and `writeBehind` so that the application can always work at its maximum speed.

This document covers information specific to using the WebLogic File API. You should also read Developing WebLogic Server Applications. If this is your first experience working with `InputStream` and `OutputStream` in Java, you may also want to read the information available in the JavaSoft tutorial.

WebLogic File API Reference

The following classes and interfaces make up the weblogic.io.common package.

Package `weblogic.io.common`
Class `java.lang.Object`
  Interface `weblogic.io.common.IOServicesDef`
  Class `java.io.InputStream`
    Class `weblogic.io.common.T3FileInputStream`
  Class `java.io.OutputStream`
    Class `weblogic.io.common.T3FileOutputStream`
  Interface `weblogic.io.common.T3File`
  Interface `weblogic.io.common.T3FileSystem`
  Class `java.lang.Throwable`
   (implements `java.io.Serializable`)
    Class `java.lang.Exception`
      Class `weblogic.common.T3Exception`

WebLogic File Objects and Classes

`weblogic.io.common.T3File`
`weblogic.io.common.T3FileSystem`
: The interfaces `T3File` and `T3FileSystem` define T3Files and T3FileSystems. T3Files, which may represent local (usually client-side) or remote (usually server-side) files, are produced by T3FileSystems, which

may also represent local or remote files. T3Files and T3FileSystems make it easy to write code that treats local and remote files uniformly. Objects from these interfaces, like all service-related objects in the WebLogic framework, are allocated by requests to an object factory. This gives the developer a fine level of control over resources.

`weblogic.io.common.T3FileOutputStream`
`weblogic.io.common.T3FileInputStream`

Two classes from the `weblogic.io.common` package, `T3FileInputStream` and `T3FileOutputStream`, provide server-side read and write access to files.

`weblogic.io.common.IOServicesDef`
`weblogic.common.T3ServicesDef`

With its class variable `services`, a WebLogic client accesses the WebLogic Server's services through methods in the `weblogic.common.T3ServicesDef`. WebLogic Files and WebLogic File Systems are accessed through the method, `T3ServicesDef.io`(), which returns a `weblogic.io.common.IOServicesDef` object.
The `IOServicesDef` interface has methods for requesting a T3FileSystem from the IOServices object factory (see "Manufacturing T3File-Related Objects"). From a client, you supply the name of the `fileSystem` as an argument to `IOServicesDef.getFileSystem`() and are returned a T3FileSystem object. From a server-side object, call `IOServicesDef.getFileSystem`() with an empty string or `null`. This returns a pointer to the file system relative to the working directory of the server.

The `T3FileSystem` interface has methods for requesting a T3File from the `IOServicesDef` object factory, and the `T3File` interface has methods for requesting a `T3FileInput/OutputStream` for reading or writing to the file.

The following code shows how a client obtains a T3FileSystem remote interface, a T3File, and an OutputStream for writing to the file:

```
T3ServicesDef t3services;
Hashtable env = new Hashtable();
env.put(Context.PROVIDER_URL, "t3://localhost:7001");
env.put(Context.INITIAL_CONTEXT_FACTORY,
          weblogic.jndi.WLInitialContextFactory.class.getName());
Context ctx = new InitialContext(env);
t3services = (T3ServicesDef)
ctx.lookup("weblogic.common.T3Services");
ctx.close();
T3FileSystem myFS = t3services.io().getFileSystem("usr");
```

```
T3File myFile = myFS.getFile("myDirectory/myFilename");
T3FileOutputStream t3os = myFile.getFileOutputStream();
t3os.write(b);
```

Wrap the code in a `try`/`catch` block to handle possible exceptions.

The recommended method of getting a `T3FileInputStream` or `T3FileOutputStream` for a T3File is to invoke `T3File.getFileInputStream`() or `T3File.getFileOutputStream`() directly on a T3File object. `T3FileInputStream` and `T3FileOutputStream` objects both extend standard `java.io.*` classes.

Setting Up the WebLogic Server to Read and Write Files

Before you can use WebLogic File services, you must first establish one or more path prefixes -- a *fileSystem* -- for use by clients. Set the Name and Path attributes for File T3 service in the Administration Console. For example, to map the file system name `users` to the path on the server host `/usr/local/tmp`, specify the Name as `users` and specify the Path as `/usr/local/tmp`. a

When you request a T3FileSystem from the `IOServicesDef` factory -- eventually to be used to creating a T3File and reading/writing to it with an input or output stream -- you use the registered `fileSystem` name as an argument for the `getFileSystem`() method. The T3FileSystem object that is returned is mapped to the specified fileSystem.

For security reasons, a WebLogic client cannot access files higher in the directory than the lowest directory registered as part of a file system name. Filenames cannot contain dot dot (`..`) or an Exception is thrown. For example, an attempt to read or write `/users/../filename` throws an Exception.

Note: When you are setting file attributes on a Windows NT system, you cannot use single backslashes (\) because they are interpreted as escape characters. Using single backslashes when setting a property result in an error message similar to this:

```
java.io.FileNotFoundException: Remote file name <filename>
malformed
```

You can either use double backslashes, as in this example:

```
weblogic.io.volume.vol=c:\\remote\\temp
```

or use forward slashes instead, which are properly mapped to a Window-style syntax by the parser:

```
weblogic.io.volume.vol=c:/remote/temp
```

Manufacturing T3File-Related Objects

In these examples, we show how to obtain request the input and output streams necessary to read and write to a remote T3File. We obtain the remote T3File object from a T3FileSystem interface. Here, `users` is the name of a fileSystem that is specified using the Administration Console. It maps to the absolute path `/usr/local/users` on the WebLogic Server host.

```
T3ServicesDef t3services = getT3Services("t3://localhost:7001");
// Get a T3FileSystem object from the IOServicesDef factory
// Give a registered fileSystem as an arg
T3FileSystem myFS = t3services.io().getFileSystem("users");
// Get a T3File from the T3FileSystem
T3File myFile = myFS.getFile("ben/notes");
// Get an OutputStream to write to the file
T3FileOutputStream t3os = myFile.getFileOutputStream();
// Write a byte "b" to the OutputStream
t3os.write(b);
```

This code creates and writes a byte to a file that maps to the Server host path `/usr/local/users/ben/notes`.

The method `getT3Services`() is in class `weblogic.common.T3Client`. You can add this method to your client.

This brief example illustrates the most common usage. There are other way to request particular T3File-related objects from the IOServicesDef factory, through a set of convenience methods that allow you to directly request a T3FileInputStream or T3FileOutputStream without first creating a T3FileSystem or T3File object.

Here are examples of using the convenience methods provided by the IOServicesDef factory.

You can request a T3FileInputStream or T3FileOutputStream object directly from the IOServicesDef factory by calling the `getFileInput/OutputStream`() method, with a *pathname* argument that follows this pattern:

`/registeredFileSystem/fileName`

where the *registeredFileSystem* is a mount-point registered in the Administration Console as the Path attribute and the *fileName* is the name of the destination file.

When you request a T3FileInputStream or T3FileOutputStream object directly, without getting one from methods called on a T3FileSystem, you must include the leading slash in the fileSystem name or the server generates this type of error:

```
java.io.FileNotFoundException: Remote file name filename is
relative
```

The T3FileInputStream object uses the defaults for buffer size and readAhead. If you choose not to use the default settings for buffer size and readAhead/writeBehind, you can set these values by using different factory methods that allow you to specify these values. In this example, an InputStream object is created with a buffer size of 1024 bytes and 3 readAhead buffers:

```
int bufferSize = 1024;
int readAhead = 3;

T3ServicesDef t3services = getT3Services("t3://localhost:7001");
InputStream is =
    t3services.io().getFileInputStream("/users/myfile",
                                       bufferSize,
                                       readAhead);
```

In this example, the OutputStream object is created with a buffer of 1024 bytes and 2 writeBehind buffers. For information on `getT3Services()`, see the javadoc for the T3Services class.

```
  int bufferSize = 1024;
  int writeBehind = 2;

  T3ServicesDef t3services = getT3Services("t3://localhost:7001");
  Outputstream os =
    t3services.io().getFileOutputStream("/users/myfile",
                                        bufferSize,
                                        writeBehind);
```

If an error occurs, the factory methods throw the exception `weblogic.common.T3Exception`, which contains the cause of the problem as a nested exception.

T3FileSystems and T3Files

`weblogic.io.common.T3FileSystem`

A T3FileSystem is made up of T3Files. You create and manage a T3File by manufacturing a T3FileInput/OutputStream that is used to read and write the file. A T3FileSystem may represent the local file system on a client, or a remote file system on a WebLogic Server. This makes it easy to write code that treats both local and remote file systems uniformly.

You request a T3FileSystem from the IOServicesDef factory with the `getFileSystem()` method. From a client, you supply the name of the fileSystem as an argument to `IOServicesDef.getFileSystem()` and are returned a T3FileSystem object. From a server-side object, call `IOServicesDef.getFileSystem()` with an empty string or `null`. This returns a pointer to the file system relative to the working directory of the server. The `T3FileSystem` interface also has other methods that return the file-system-dependent file separator string, and the file-system-dependent path separator string. This interface also contains more convenience methods that allow direct access to file Input/OutputStreams without creating an intermediary T3File object.

`weblogic.io.common.T3File`

You request a T3File by calling one of the `T3FileSystem.getFile()` methods. Like a T3FileSystem, a T3File can represent either local or remote files. In addition to methods for getting Input/OutputStreams to read and write to the file, this interface also has accessory methods to get the file name and path associated with the T3File object, to get its parent directory, to check if the file exists and is a normal T3file, to test if you can read and write to the file, to get its length and last modified date, to rename it, to make a directory, and other file-related tasks.

The T3FileInputStream Class

`weblogic.io.common.T3FileInputStream`

You customarily create a T3FileInputStream by calling the `T3File.getFileInputStream()` method, which returns an object of the class `T3FileInputStream`. This class extends the standard `java.io.InputStream` class and provides two additional methods:

```
public int bufferSize();
```

which returns the current buffer size and

```
public int readAhead();
```

which returns the current number of buffers of read ahead.

The implementation of two other methods in `T3FileInputStream` that override methods in `java.io.InputStream` are of interest:

- The method `available`() returns the number of bytes of unread data that have been buffered on the client. It is never greater than the buffer size times one plus the number of buffers of read ahead.
- The method `skip`() starts out by discarding data that has been requested through read ahead and eventually issues a request to the server to skip any remaining data.

Currently, T3FileInputStream does not support the following methods: `java.io.InputStream.mark`() and `java.io.InputStream.reset`().

The T3FileOutputStream Class

`weblogic.io.common.T3FileOutputStream`

You customarily create a T3FileOutputStream by calling the `T3File.getFileOutputStream`() method, which returns an object of class `T3FileOutputStream`. This class extends the standard `java.io.OutputStream` class and provides two additional methods:

```
public int bufferSize();
```

which returns the current buffer size and

```
public int writeBehind();
```

which returns the current number of buffers of write behind. The implementation of two other methods in `T3FileOutputStream` that override methods in `java.io.OutputStream` are of interest:

- The method `flush`() blocks on the client until an acknowledgment has been received that all outstanding buffers have been flushed to the server.
- The method `close`() method does an automatic `flush`().

If an error occurs on the server while a file is being written, the client is asynchronously notified and all subsequent operations -- `write()`, `flush()`, or `close()` -- generates a `java.io.IOException`.

Programming with WebLogic File Services

Here are step-by-step instructions on how to request and use T3File-related objects in your application.

- Step 1. Import Packages
- Step 2. Obtain a Remote T3Services Interface
- Step 3. Create a T3FileSystem and a T3File
- Step 4. Create and Use an OutputStream Object
- Step 5. Create and Use an InputStream Object

A code example is provided following these steps.

Step 1. Import Packages

In addition to other packages you import for your program, WebLogic File applications import the following:

```
import java.io.*;
import weblogic.common.*;
import weblogic.io.common.*;
```

Step 2. Obtain a Remote T3Services Interface

From a WebLogic client application, you access the T3File services via the T3ServicesDef remote factory interface that lives on the WebLogic Server. Your client obtains a remote stub to the T3Services object via a JNDI look-up. We define and list

a method called `getT3Services()` that you can add to you client to access the T3Services stub. For information on `getT3Services()`, see the javadoc for the T3Services class.

You can simply call the method giving the URL of the WebLogic Server as an argument as follows:

```
T3ServicesDef t3services = getT3Services("t3://weblogicurl:7001")
```

Step 3. Create a T3FileSystem and a T3File

In general, perform the following steps to begin working with reading and writing files:

- Get a T3FileSystem object.
- Make a request to the T3FileSystem object for a T3File. You can then read from or write to this file.

Use the `T3ServicesDef` remote interface to access the IOServices factory. Call the IOServices factory method `getFileSystem()` to get a `T3FileSystem` object. Supply the name of a file system that is registered on the WebLogic Server as an argument. You register a file system using the Administration Console.

For this example, we assume the following file system property is configured with a name of `myFS` and a path of `/usr/local`.

T3Files created in the T3FileSystem mapped to `myFS` are physically located in the directory `/usr/local` on the WebLogic Server's host. Here is the code to get the T3FileSystem and a T3File named `test`:

```
T3FileSystem t3fs =
  t3services.io().getFileSystem("myFS");
  T3File myFile = t3fs.getFile("test");
```

We can also check to see if the file exists before we read from or write to it, as shown here:

```
if (myFile.exists()) {
  System.out.println("The file already exists");
}
else {
  // Create a file with an array of bytes. We'll write it
  // to an output stream in the next step
```

```
    byte b[] = new byte[11];
    b[0]='H'; b[1]='e'; b[2]='l'; b[3]='l'; b[4]='o'; b[5]=' ';
    b[6]='W'; b[7]='o'; b[8]='r'; b[9]='l'; b[10]='d';
  }
```

Step 4. Create and Use an OutputStream Object

We have assembled an array of bytes that we'd like to write to a T3File on the WebLogic Server. You customarily create a T3File, and then request an `OutputStream` to write to it, using the `T3File.getOutputStream`() method.

Using the T3File `myFile` that we created in the previous step, this example illustrates this process:

```
Outputstream os =
   myFile.getFileOutputStream();
   os.write(b);
   os.close();
```

Always close the `OutputStream` object when work with it is completed.

Step 5. Create and Use an InputStream Object

Now we have got a T3File that we'd like to read from and confirm its contents. You request and use an `InputStream` object with the same patterns you use for an `OutputStream` object.

Here we request an `InputStream` object with which we can read from the T3File `myFile`. This opens an `InputStream` to the T3File. In this example, we're reading bytes; first, we allocate an array of bytes to read into. This array is later used to create a String that can be displayed. Then, we use the standard methods of the `java.io.InputStream` class to read from the T3File, as shown here:

```
byte b[] = new byte[11];
InputStream is = myFile.getFileInputStream();
is.read(b);
is.close();
```

Now let's create a String for display to confirm the results:

```
String result = new String(b);
System.out.println("Read from file " + T3File.getName()
                   " on the WebLogic Server:");
System.out.println(result);
is.close();
```

Always close the `InputStream` object when work with it is completed.

Code Example

The full code example is a runnable example that we ship in the `examples/io` directory in the distribution. You can compile and run the example using the instructions located in the same directory. The example uses a `main()` method so that you can run the example from the command line.

```
public class HelloWorld {

  public static void main(String[] argv) {

    // Strings for the WebLogic Server URL, the T3FileSystem
    // name, and the T3File name
    String url;
    String fileSystemName;
    String fileName;

    // Check the user's input, and then use it if correct
    if (argv.length == 2) {
      url = argv[0];
      // Use the local file system on the client
      fileSystemName = "";
      fileName = argv[1];
    }
    else if (argv.length == 3) {
      url = argv[0];
      fileSystemName = argv[1];
      fileName = argv[2];
    }
    else {
      System.out.println("Usage:   java example.io.HelloWorld " +
                         "WebLogicURL fileSystemName fileName");
      System.out.println("Example: java example.io.HelloWorld " +
                         "t3://localhost:7001 users test");
      return;
    }
```

```
    // Obtain remote T3Services factory from WebLogic Server
    try {
      T3Services t3services = getT3Services(url);

      // Get the file system and the file
      System.out.println("Getting the file system " + fileSystemName);
      T3FileSystem fileSystem =
        t3services.io().getFileSystem(fileSystemName);
      System.out.println("Getting the file " + fileName);
      T3File file = fileSystem.getFile(fileName);

      if (file.exists()) {
// The file exists. Don't do anything
System.out.println("The file already exists");
      }
      else {
// The file does not exist. Create it.
byte b[] = new byte[11];
b[0]='H'; b[1]='e'; b[2]='l'; b[3]='l'; b[4]='o'; b[5]=' ';
b[6]='W'; b[7]='o'; b[8]='r'; b[9]='l'; b[10]='d';

// Get an OutputStream and write to the file
System.out.println("Writing to the file");
OutputStream os = file.getFileOutputStream();
os.write(b);
os.close();
      }

      // Get an InputStream and read from the file
      byte b[] = new byte[11];
      System.out.println("Reading from the file");
      InputStream is = file.getFileInputStream();
      is.read(b);
      is.close();

      // Report the result
      String result = new String(b);
      System.out.println("File contents is: " + result);
    }
    catch (Exception e) {
      System.out.println("The following exception occurred " +
                         "while running the HelloWorld example.");
      e.printStackTrace();
      if (!fileSystemName.equals("")) {
System.out.println("Make sure the WebLogic server at " +
                   url + " was started with " +
                   "the property weblogic.io.fileSystem." +
                   fileSystemName + " set.");
      }
```

```
    }
  }

  private static T3ServicesDef getT3Services(String wlUrl)
      throws javax.naming.NamingException
  {
    T3ServicesDef t3s;
    Hashtable env = new Hashtable();
    env.put(Context.PROVIDER_URL, wlUrl);
    env.put(Context.INITIAL_CONTEXT_FACTORY,
            weblogic.jndi.WLInitialContextFactory.class.getName());
    Context ctx = new InitialContext(env);
    t3s = (T3ServicesDef) ctx.lookup("weblogic.common.T3Services");
    ctx.close();
    return(t3s);
  }
}
```