WebSphere Application Server for z/OS Version 7

The WOLA Native APIs ... a COBOL Primer

A series of structured exercises, from simple to increasingly advanced, illustrating the WOLA native APIs ... both inbound and outbound.

Version Date: August 13, 2013

See "Document Change History" on page 57 for a description of the changes in this version of the document

IBM Advanced Technical Skills
Gaithersburg, MD

WP101490 at
ibm.com/support/techdocs

Many, *many* thanks to **Jim Mulvey**, **Tim Kaczynski** and **Dave Follis** of the WAS z/OS development team. And a special thanks to **Leigh Compton** and **Dennis Weiand** of ATS for their help with the more complex COBOL coding issues.

And of course the IBM customers who are using WOLA!

The WAS z/OS support team in IBM Advanced Technical Skills consists of **John Hutchinson**, **Mike Kearney**, **Louis Wilen**, **Lee-Win Tai**[1], **Mike Loos**, **Paul Houde** and **Don Bagwell**.

We also receive wonderful support from **Dennis McDonald** and **Brian Pierce**.

Mike Cox, Distinguished Engineer, serves as technical advisor to all our activities.

1 This paper owes much to Lee-Win's Java skills, particularly as it relates to WOLA.

Table of Contents

Introduction and Overview

This document is intended to assist you in becoming comfortable with coding to the WOLA native APIs. The document is provided in a primer format -- "A book that covers the basic elements of a subject." [2]

We'll do this by providing structured lessons that start simple then layer up to more complex topics and issues.

To get the most out of this document you'll need access to a z/OS system that has the following:

- WebSphere Application Server for z/OS V7.0.0.4 or higher
- A configured runtime environment, either Network Deployment or Standalone Server
- A node in that runtime environment enabled for WOLA support

> **Note:** In the WAS z/OS InfoCenter[3], search on `tdat_enableconnector`. That will take you to the page that provides the step-by-step instructions to do this. We provide a quick checklist of things under "Quick checklist for enabling the WAS environment for these exercises" on page 54.

Why COBOL?

Because it's a very prevalent z/OS programming language, and it may be the most common language used with CICS.

Truth is, the API usage is essentially the same between COBOL and C/C++ with the exception of some syntax differences. *Functionally* it's the same, and the parameters are the same, but there are differences in the way certain formatting and delimiting characters are coded.

Java skills needed?

Not for the exercises in this document. The Java sample program supplied with WOLA will be used for all these exercises. That sample Java program makes use of the methods implemented on the WOLA JCA resource adapter that's shipped with the function.

What about CICS and IMS?

Both CICS and IMS support use of WOLA to communicate with WAS. CICS support was part of the original offering of 7.0.0.4; IMS support came into play with 7.0.0.12.

Both CICS and IMS provide a way to "shield" your external programs from having to use the APIs. CICS provides a "Link Server Task" which invokes the target program using EXEC CICS LINK. IMS provides a way to shield the program behind the OTMA interface.

> **Note:** This document will not go into the specifics of enabling the CICS Link Server Task or the IMS OTMA support. The InfoCenter has detailed instructions on that.

The message is this -- the exercises in this document do not *require* CICS or IMS. Batch COBOL with a WOLA-enabled WAS z/OS server is all you need.

Ready? Let's begin ...

2 dictionary.com
3 http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp

Important Sources of Information

InfoCenter

http://publib.boulder.ibm.com/infocenter/wasinfo/v7r0/index.jsp

z/OS

Network Deployment (z/OS), Version 7.0

View the latest Network Deployment and single server z/OS® documentation. This information applies to Version 7.0 and to all subsequent releases and modifications until otherwise indicated in new editions.

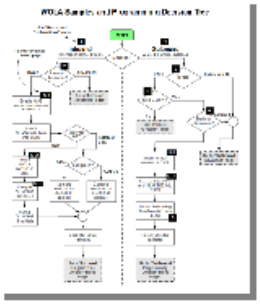
- Latest updates to the information center - Read the latest on changes made to the information and the frequency of information center updates.

IBM Techdocs

http://www.ibm.com/support/techdocs

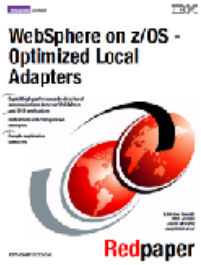
WP101490 - WebSphere z/OS Optimized Local Adapters

- **This document**
- **Overview presentations**
- **"Planning and Design Guide"**



IBM Redbooks and Redpieces

http://www.redbooks.ibm.com/



REDP4550

A comprehensive look at WOLA soup-to-nuts, from installation to development of solutions.

WOLA on YouTube!

http://www.youtube.com/user/WASOLA1

Demos and narration from WOLA developers!

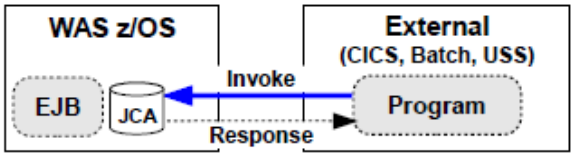
Please consult each resource according to the requirements of your solution project

Some Essential WOLA Level-Setting

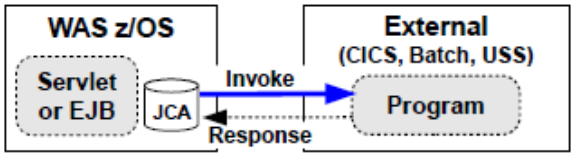
First a little preliminary education to set the stage.

Inbound vs. Outbound

The key is *who initiates the call*, the Java program or the COBOL program? The following picture illustrates this concept:



Inbound Model



Outbound Model

Some of the APIs relate to one, and some of the APIs relate to the other. That's why having this distinction in mind is important.

Native APIs vs. Java APIs

The focus of this document is the native APIs. By that we mean the ones used by the *non-Java* languages such as COBOL.

There *are* Java APIs associated with WOLA[4]. They are provided in the JCA resource adapter mentioned earlier. That resource adapter, called `ola.rar`, implements the Common Client Interface (CCI).

The Native APIs

There are 13 native APIs listed in the InfoCenter[5]:

- Register - BBOA1REG
- Unregister - BBOA1URG
- Connection Get - BBOA1CNG
- Connection Release - BBOA1CNR
- Send Request - BBOA1SRQ
- Send Response - BBOA1SRP
- Send Response Exception - BBOA1SRX
- Receive Request Any - BBOA1RCA
- Receive Request Specific - BBOA1RCS
- Receive Response Length - BBOA1RCL
- Get Message Data - BBOA1GET
- Invoke - BBOA1INV
- Host Service - BBOA1SRV

The InfoCenter page offers a wonderful reference of each, including the parameter syntax, parameter data typing, return codes and reason codes. For example, the `BBOA1REG` API's parameter syntax looks like this:

Table 1. BBOA1REG API syntax. The syntax is explained in the Parameters section.

API	Syntax
BBOA1REG	`BBOA1REG ( daemongroupname , nodename , servername , registername , minconn , maxconn , registerflags , rc, rsn )`

But that begs the questions, "What does `BBOA1REG` *do*?" and "What are all those parameters?" and what are all those *other* APIs?

And that's what we'll cover throughout this document.

4 See the "Design and Planning Guide" PDF associated with the WP101490 Techdoc.
5 Search on `cdat_olaapis`

Some essential concepts

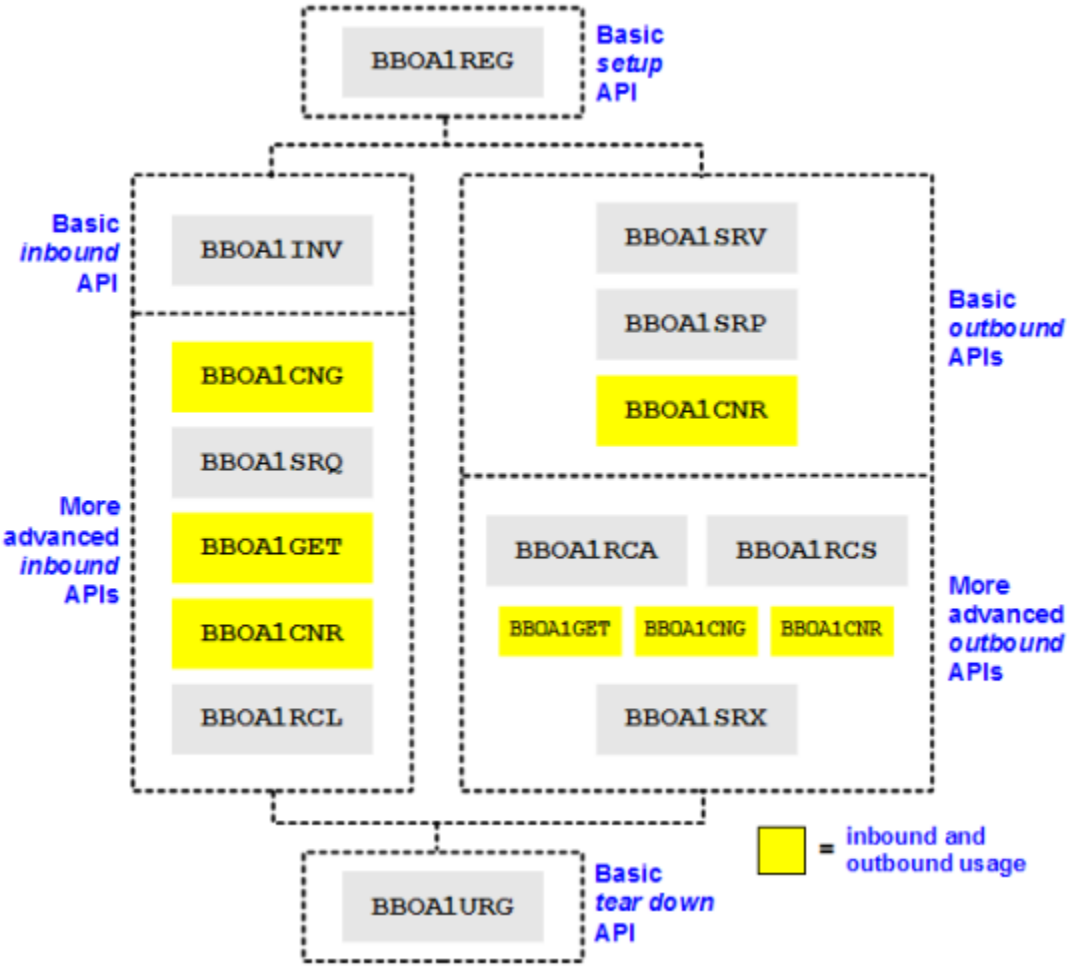
Terms or Phrases	*Explanation*
Register **Registration** **Registering**	This is the first and most fundamental type of WOLA link between your external program and an application server. Think of it as the basic "pipe" used between WAS and the external address space. Multiple registrations is possible. Registering establishes a pool of shared memory that is used for control information and the exchange of messages[6] over WOLA. It also establishes a connection pool within the registration. More on that in a bit. The external program *always* initiates the registration. WAS receives the request and establishes the linkage. Registrations carry a name and other settings. When you wish to send or receive a message you have to indicate which register name to use.
Unregister **Unregistration** **Unregistering**	The opposite of registration. Unregistering tears down (removes) the connection pool and the shared memory allocation. The registration name is removed from the list maintained by the WAS cell. The external program initiates unregistration. WAS receives the request and processes the removal and cleanup of the registration.
Requests	Requests are what goes from originator to the target.
Responses	Responses are what comes back from the target.
Connections	Within a particular registration there exists some number of *connections*, which are maintained in a pool. The minimum and maximum number of connections is specified at the time of registration. Requests and responses flow over a connection. Multiple connections within a registration allows multi-threading of requests and responses.
Invoke (or Send)	Originators invoke (or send) a request.
Host (or Receive)	As we mentioned, an originator invokes (or sends) a request. That means *something* must be on the other side to receive and act on the request. When going inbound to WAS from an external address space, that *something* is the WOLA support code inside WebSphere Application Server. You don't really see this activity; it's hidden under the covers. When going outbound from WAS to an external address space, that *something* is your program[7]. "Hosting" a service is the act of your program using one of the WOLA APIs to put itself into a state of readiness in anticipation of a message. The WP101490 "Design and Planning Guide" categorized this as "advanced." It's not really that complicated, but it's more complicated than a simple invoke. "Receive" is a more granular type of "hosting a service."
Service	A request is sent over a *connection*, which is a subset of the *registration* pool. A service is a name that represents the final target. Originators need to know the name of the target so it can specify who to deliver the request to. In WAS the target is an EJB, and the service name is the JNDI name of the EJB. In CICS the target is a program, and the service name is the name of the program. For batch it's the service name specified on the "host a service" API.

6 Actually only very large messages pass through this shared "above the bar" shared memory; smaller messages use another cross-memory path. But that's a detail that simply doesn't matter in the context of this document.

7 The exception is when going from WAS to CICS and you use the WOLA BBO$ LINK server task. That hides the details as well. The "Design and Planning Guide" PDF of WP101490 explains how this element of WOLA works. For the purposes of this document focus on the description above.

Categorizing the APIs according to their usage

The 13 native APIs we mentioned earlier can be catgorized in this way:



What those APIs do ... in simple terms

BBOA1REG	Registers into the Daemon Group shared memory
BBOA1INV	Sends a request to WAS
BBO1ACNG	Gets a connection from the connection pool
BBOA1SRQ	Sends a request into WAS
BBOA1GET	Gets a reponse off the message thread
BBOA1CNR	Releases the connection and returns it to the pool
BBOA1RCL	Gets the length of the response so it can be pulled in and worked on
BBOA1SRV	Sets up a "host a service" function in your program
BBOA1SRP	Sends a response back to whatever came to your program
BBOA1RCA	A finer-grained version of `BBOA1SRV` -- "Receive Connection Any"
BBOA1RCS	Receives message on a specific connection using a connection handle
BBOA1SRX	Sends an exception response back if something goes wrong
BBOA1URG	Opposite of REG ... tears down the registration

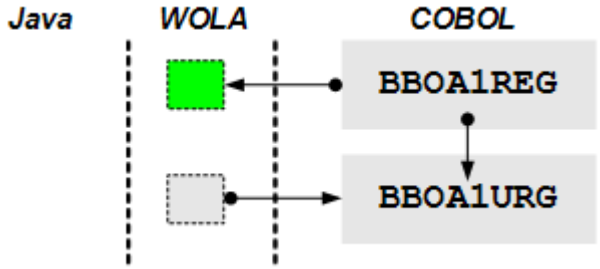
And with that we're ready to dig deeper into these APIs and their usage.

An Picture Overview Map of the Exercises in this Document

This section provides a very high-level picture representation of each exercise with a page pointer to where the specifics are covered in more detail.

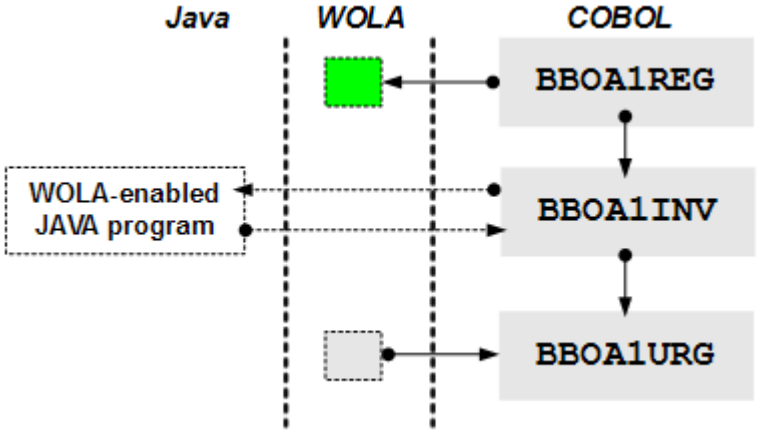
Exercise 1a -- page 18

A simple "register / unregister" exercise:



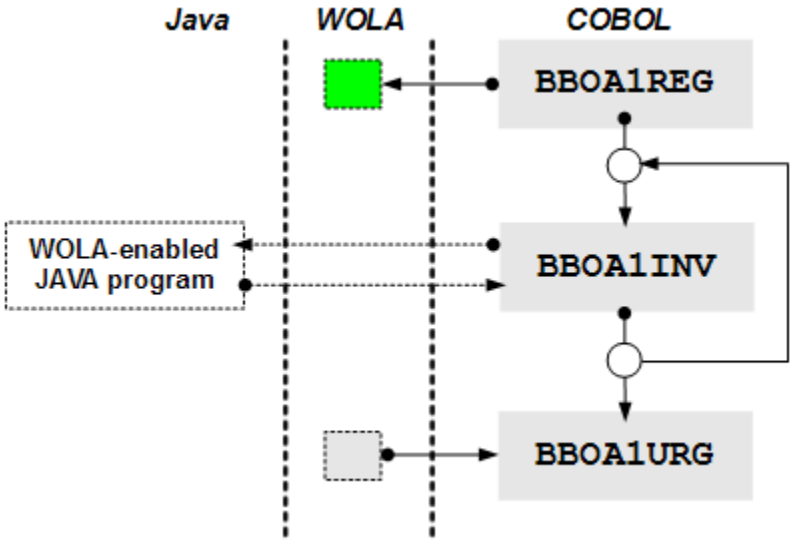
Inbound Exercise 2a -- page 23

A single invocation of the Java program:



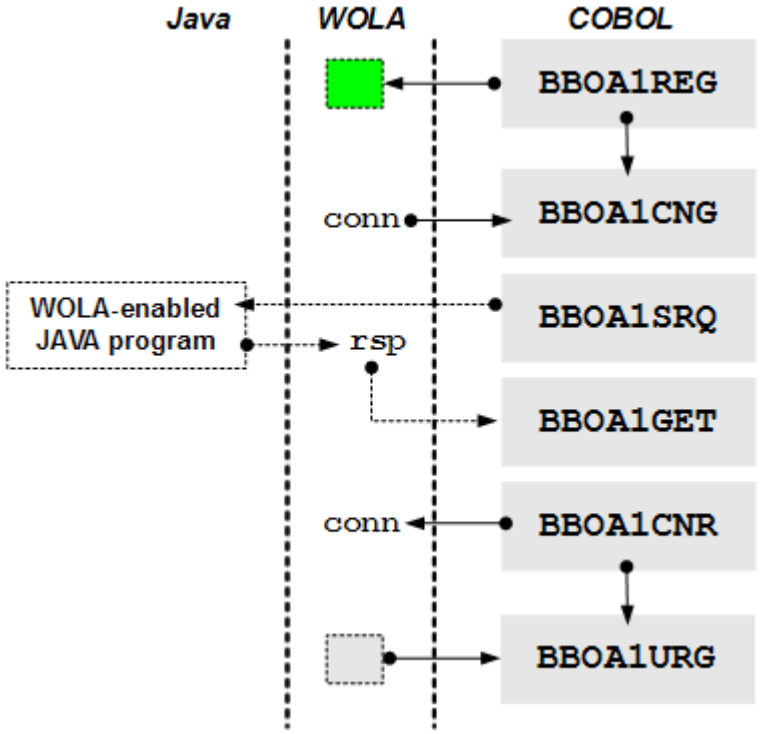
Inbound Exercise 2b -- page 26

Looping `BBOA1INV`:



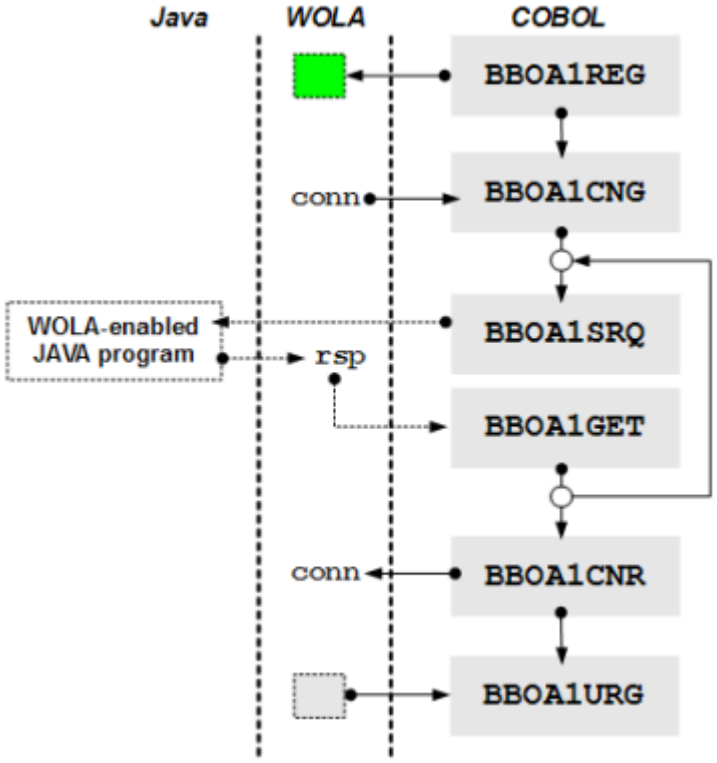
Inbound Exercise 2c -- page 29

Finer control: `BBOA1CNG`, `BBOA1SRQ`, `BBOA1GET`, `BBOA1CNR`



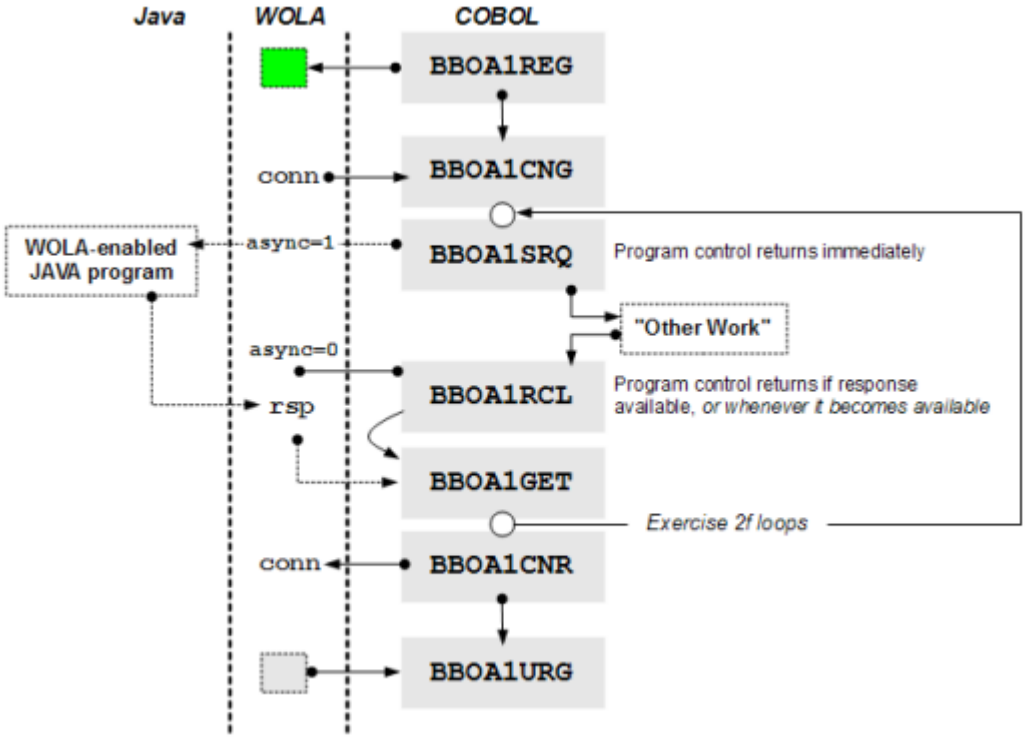
Inbound Exercise 2d -- page 32

Insert loop into Exercise 2c:



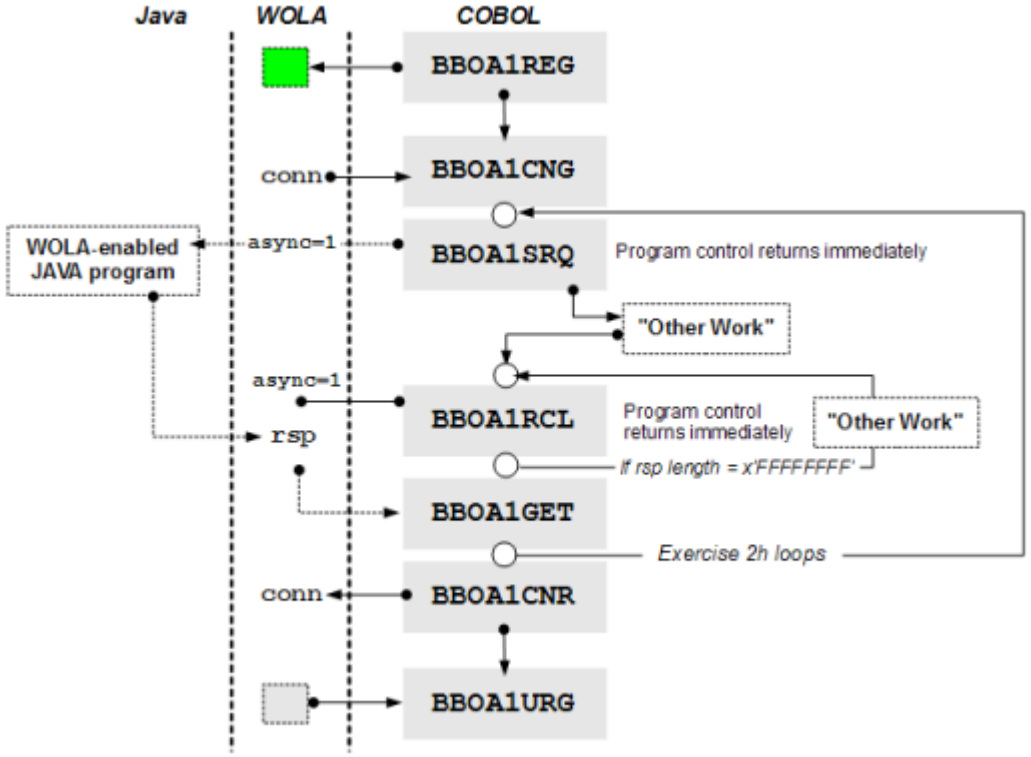
Inbound Exercise 2e and 2f -- page 34

Asynchronous `BBOA1SRQ`, "other work" and then synchronous `BBOA1RCL` (2e is single pass) and then a loop (2f is with a loop):



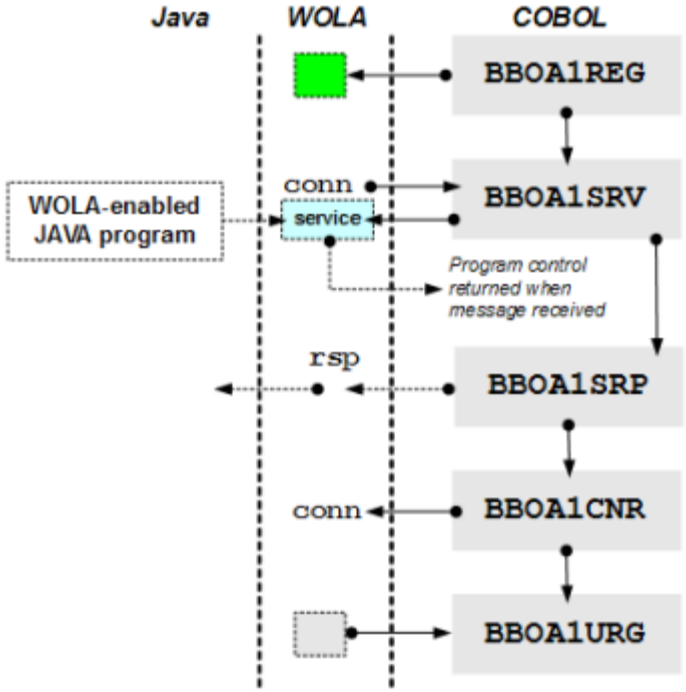
Inbound Exercise 2g and 2h -- pages 36 and 40 respectively

Asynchronous `BBOA1SRQ`, "other work" and then asynchronous `BBOA1RCL` (2g is single pass) and then a loop (2h is with a loop):



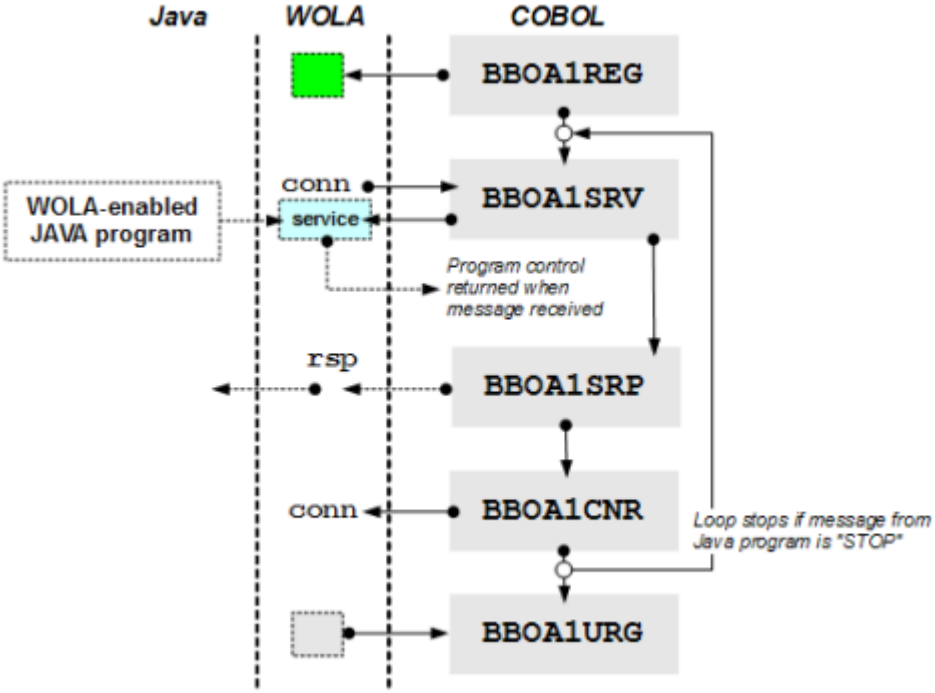
Outbound Exercise 3a -- page 44

Single pass design. Register, `BBOA1SRV` to host a service, send a response, release connection and unregister.



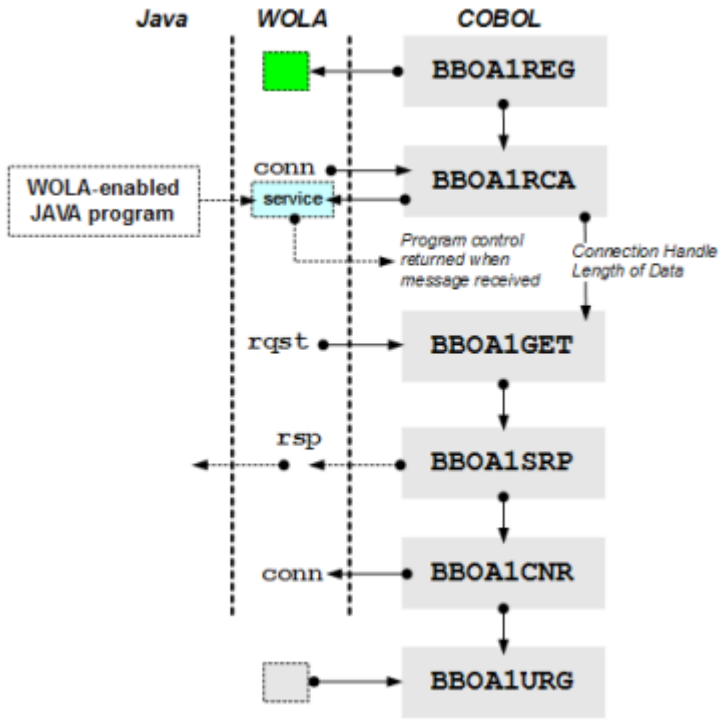
Outbound Exercise 3b -- page 50

Exercise 3a with a loop.



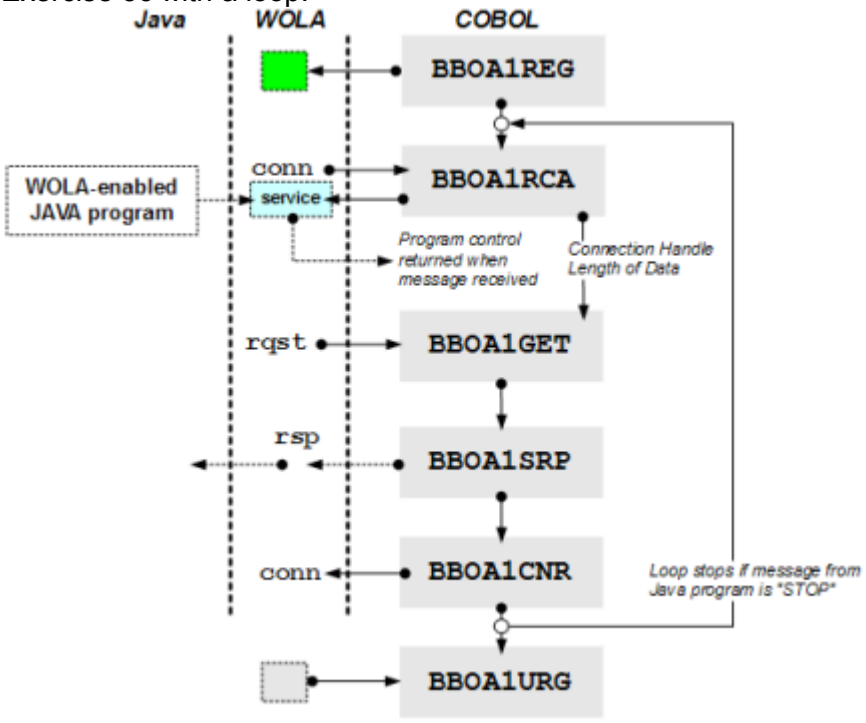
Outbound Exercise 3c -- page 52

Single pass with `BBOA1RCA` and `BBOA1GET`:



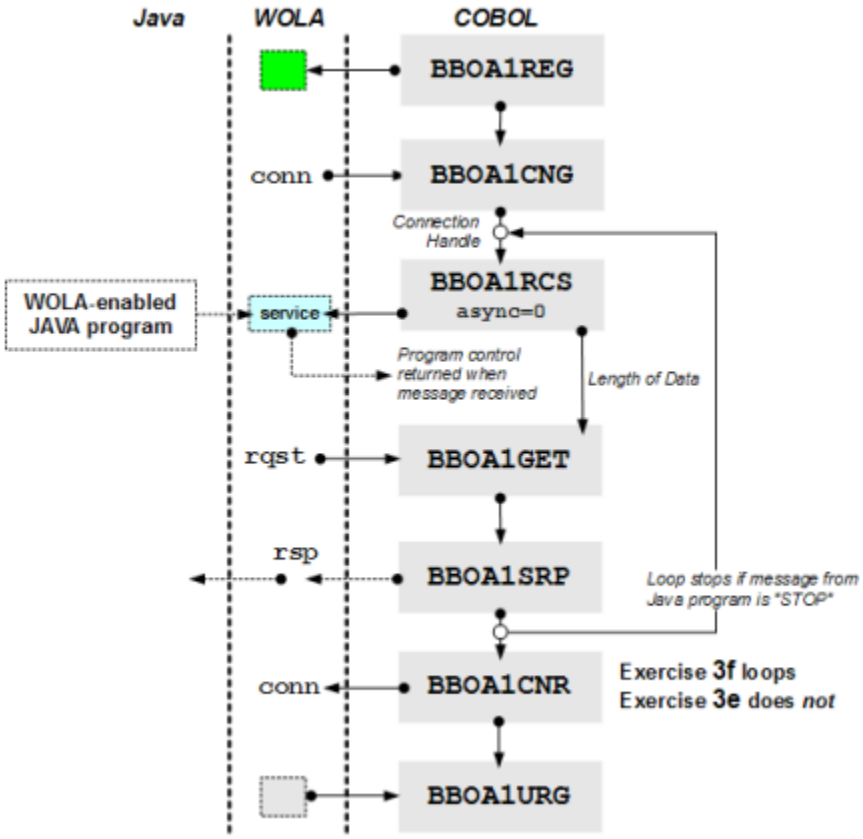
Outbound Exercise 3d -- page 53

Exercise 3c with a loop:



Outbound Exercise 3e and 3f -- page 53

`BBOA1RCS` with `async=0`[8]



Asynchronous RCS? See "What about asynchronous BBOA1RCS?" on page 53.

8 Note that the `BBOA1CNG` and `BBOA1CNR` are now *outside* the loop. We can do that because `BBOA1RCS` does not itself do a `CNG` under the covers while `BBOA1INV` and `BBOA1RCA` does. Because they issue a BBOA1CNG each time, if we didn't release the connection the program would simply get one more each loop until the maximum connections was exhausted. The maximum connections is set on `BBOA1REG`.

Unit 1: Exploring the BBOA1REG and BBOA1URG APIs

Think of BBOA1REG and BBOA1URG as the start and stop of a broad WOLA "cycle." The BBOA1REG API (simply REG for short), constructs the registration shared space, establishes the control blocks and prepares the environment to pass traffic back and forth. BBOA1URG (URG for short) does the reverse.

Before you can do anything with WOLA you have to process a REG.

Before you quit for the day you should process an URG[9] to clean up your environment.

A quick review of the BBOA1REG API

The BBOA1REG API is what establishes the initial registration into the WAS z/OS environment.

The InfoCenter (search: cdat_olaapis) provides the following syntax guide. We've highlighted the input parameters and output values:

Table 1. BBOA1REG API syntax. The syntax is explained in the Parameters section.

API	Syntax
BBOA1REG	BBOA1REG (daemongroupname , nodename , servername , registername , minconn , maxconn , registerflags , rc, rsn)

Input parameters

Output values

The key *input* parameters are the cell, node and server short names where the target EJB resides:

Table 1. BBOA1REG API syntax. The syntax is explained in the Parameters section.

API	Syntax 1 2 3
BBOA1REG	BBOA1REG (daemongroupname , nodename , servername , registername , minconn , maxconn , registerflags , rc, rsn)

Notes:

1. **daemongroupname** -- this is the cell ***short*** name for the cell into which you wish to establish the registration relationship.
2. **nodename** -- the node ***short*** name where the target server resides.
3. **servername** -- the server ***short*** name where the target server resides.

The other *input* parameters are:

Table 1. BBOA1REG API syntax. The syntax is explained in the Parameters section.

API	Syntax
BBOA1REG	BBOA1REG (daemongroupname , nodename , servername , registername , minconn , maxconn , registerflags , rc, 1) 2 3 4

Notes:

1. **registername** -- this is an arbitrary name to uniquely identify the registration. This is used in other APIs to indicate which registration to use to send the traffic. It must be exactly 12 characters long. Blank padded works.
2. **minconn** -- the starting connections in the connection pool for this registration.. The default is 1.

9 Strictly speaking you could just close down your batch program or CICS region. But processing a BBOA1URG to unregister is better practice.

3. **`maxconn`** -- tthe maximum connections in the connection pool for this registration. The default is 10[10].
4. **`registerflags`** -- this is a three-byte (32-bit) field used to pass in information about the transaction and security attributes to apply to this registration:
   - Bit 29 -- CICS to WAS identity propagation (CICS task identity)
   - Bit 30 -- CICS to WAS transaction propagation
   - Bit 31 -- WAS to CICS transaction propegation (at present fixed at `0`)

Here's an *example* of a COBOL snippet that would perform the registration:

```
WORKING-STORAGE SECTION.
01 daemongroup                  PIC X(8) VALUE LOW-VALUES.  1
01 node-name                    PIC X(8).
01 server-name                  PIC X(8).
01 register-name                PIC X(12) VALUE SPACES.
01 minconn                      PIC 9(8) COMP VALUE 1.
01 maxconn                      PIC 9(8) COMP VALUE 10.     2
01 regopts                      PIC 9(8) COMP VALUE 0.      3
01 rc                           PIC 9(8) COMP VALUE 0.
01 rsn                          PIC 9(8) COMP VALUE 0.
  :
MOVE 'OLAINBND'                     TO register-name.
MOVE 'S1CELL'                       TO daemongroup.
MOVE 'S1NODEC'                      TO node-name.           4
MOVE 'S1SR01C'                      TO server-name.
  :
INSPECT daemongroup CONVERTING ' ' to LOW-VALUES.           5
  :
CALL 'BBOA1REG' USING
      daemongroup,
      node-name,
      server-name,
      register-name,                                        6
      minconn,
      maxconn,
      regopts,
      rc,
      rsn.
 :
IF rc > 0 THEN
  DISPLAY "OLA - BBOA1REG problem -- rc/rsn : " rc "/" rsn
  GO TO Bad-RC
END-IF.
```

Notes:

1. The `daemongroup` name must be null-terminated. That's why "value low-values."
2. The values `minconn` and `maxconn` are set to default 1 and 10.
3. The `regopts` parameter is set to 32 bits of 0, which means no security propagation and no transaction propagation. Good enough for simple validation.
4. String values are populated.
5. The `daemongroup` value (cell short name) must be null-terminated. This is simply insuring that is the case.
6. The `BBOA1REG` API is called with the values input parameters populated based on earlier setting.

That's enough background information. Let's do some actual practice runs.

10 See "Design and Planning Guide" under the WP101490 Techdoc for a discussion of the performance considerations for minconn and maxconn. When first starting out the defaults are fine.

Exercise Preparation

In prep for the upcoming exercises, make certain you have:

- ☐ Access to a COBOL compiler and system link editor
- ☐ The WOLA modules copied out[11] to a load library you can reference from LKED SYSLIB
- ☐ A WAS z/OS server environment created and operational and enabled for WOLA[12].

Summary:
- Cell-level environment variable `WAS_DAEMON_ONLY_enable_adapter` = `true`
- The `ola.rar` JCA adapter installed into the target node
- The `OLASample1.ear` sample application installed into the target server
- The `CB.BIND.<prefix>.**` profile updated to provide `READ` to the ID of the batch program
- The WAS cell (including daemon) stopped and restarted to pick up the changes made

Overview of Exercise 1a - Simple `BBOA1REG` and `BBOA1URG`

Note: The source for this exercise and all others can be found in the ZIP file that accompanies this document under the WP101490 Techdoc at `ibm.com/support/techdocs`.

This exercise performs a very simple registration and unregistration from the specified WAS z/OS "daemon group." It'll execute very quickly as it does no work beyond that.

The exercise code looks like this:

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER1A.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      * API Parms
       01 daemongroup        PIC X(8) VALUE LOW-VALUES.
       01 node-name          PIC X(8).
       01 server-name        PIC X(8).
       01 register-name      PIC X(12) VALUE SPACES.
  1    01 minconn            PIC 9(8) COMP VALUE 1.
       01 maxconn            PIC 9(8) COMP VALUE 10.
       01 regopts            PIC 9(8) COMP VALUE 0.
       01 urgopts            PIC 9(8) COMP VALUE 0.
       01 rc                 PIC 9(8) COMP VALUE 0.
       01 rsn                PIC 9(8) COMP VALUE 0.
      * Procedures Section
       PROCEDURE DIVISION.
       MAINLINE SECTION.
           MOVE 'EXER1A'     TO register-name.
           MOVE 'S1CELL'     TO daemongroup.
  2        MOVE 'S1NODEC'    TO node-name.
           MOVE 'S1SR01C'    TO server-name.

  3        INSPECT daemongroup CONVERTING ' ' to LOW-VALUES.

           CALL 'BBOA1REG' USING
                 daemongroup,
                 node-name,
  4              server-name,
                 register-name,
                 minconn,
                 maxconn,
                 regopts,
                 rc,
```

11 The `olaInstall.sh` does that for you. InfoCenter search: `tdat_enableconnector` for the syntax.

12 InfoCenter: `tdat_enableconnector`

```
                rsn.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1REG problem -- rc/rsn : " rc "/" rsn
5          GO TO Bad-RC
           ELSE
             DISPLAY "Successfully registered into " daemongroup
           END-IF.

           CALL 'BBOA1URG' USING
6              register-name,
               urgopts,
               rc,
               rsn.

           IF rc > 0 THEN
7            DISPLAY "OLA - BBOA1URG problem -- rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           ELSE
             DISPLAY "Successfully unregistered from " daemongroup
           END-IF.

           GOBACK.

       *  Exit with bad-RC

8       Bad-RC.
           DISPLAY "OLA - EXITING program due to non-RC=0."
           GOBACK.
/*
```

Notes:

1. Setting up the working storage definitions. The InfoCenter (search: `cdat_olaapis`) has information on each of the API parameters. Notice how some have initial values supplied, such as `minconn`, `maxconn` and the options fields.
2. The key information is supplied for the registration -- register name (must be exactly 12 characters[13], must be unique from any other registrations in place, but otherwise arbitrary), and the *short* names: cell (called "daemongroup"), node and server.
3. The daemon group name must be null terminated, which is why this `INSPECT` is used.
4. The `BBOA1REG` API is called and the variables are passed in.
5. Simple `IF` structure to check for RC and report on what it sees.
6. The `BBOA1URG` API is called with its parameters passed in.
7. Another simple `IF`.
8. Where program goes if `RC>0`.

The output in `SYSOUT` should looks like this:

```
Successfully registered into S1CELL
Successfully unregistered from S1CELL
```

Perform Exercise 1a

Do the following:

☐ Allocate a target module library where the compiled module will go.

13 Because the working storage definition was `PIC X(12) VALUE SPACES` is becomes blank padded, which is acceptable.

- ☐ Copy the supplied sample `exer1a.txt` to your system. Update the compile and link edit values according to your system.
- ☐ Update the `daemongroup`, `node-name` and `server-name` values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the `runprog.txt` JCL supplied in the Techdoc ZIP file.
- ☐ Check the `SYSOUT` ... what RC did you get from `BBOA1REG`?
  - `RC=0, RSN=0` Successful registration
  - `RC=12, RSN=10` Daemon group not found ... typo or WAS not started[14]
  - `RC=12, RSN=16` Node or server name not found ... typo, or the target application server wasn't started
  - `RC=12, RSN=24` Daemon up, but server named is not.
  - `RC=12, RSN=30` "The daemon group is not running with `WAS_DAEMON_ONLY_enable_adapter` property set to 1." [15]

Note: If you received any non-0 RC then go back and correct the problem, recompile and re-run until you get a successful register/unregister.

Variations on Exercise 1a - forced error conditions

In this exercise you create some intentional error conditions to see the RC/RSN thrown and match that against what's in the InfoCenter. Make the change indicated, recompile and re-run.

`BBOA1REG` errors:

- ☐ Provide an incorrect `daemongroup` -- should get `RC=12/RSN=10`
- ☐ Provide an incorrect `node-name` or `server-name` -- should get `RC=12/RSN=16`
- ☐ Stop the application server but leave the Daemon up -- should get `RC=12/RSN=24`
- ☐ Duplicate the block of `BBOA1REG` code, including the `IF-THEN` structure, so the program will try to register twice with the same name. Should get `RC=8/RSN=8`[16]
- ☐ Restore the program back to its working `RC=0/RC=0` state.

`BBOA1URG` errors:

- ☐ Copy the line: `MOVE 'EXER1A' TO register-name` to just *above* the `BBOA1`**`URG`** call.
- ☐ On the new copy of the line, change the registration name to something *other than* what was used earlier to register. That creates a condition where you're trying to unregister using a name that's not currently in the registration list.

  You should see `RC=8/RSN=8`, which the InfoCenter indicates means "Registration token name does not exist."

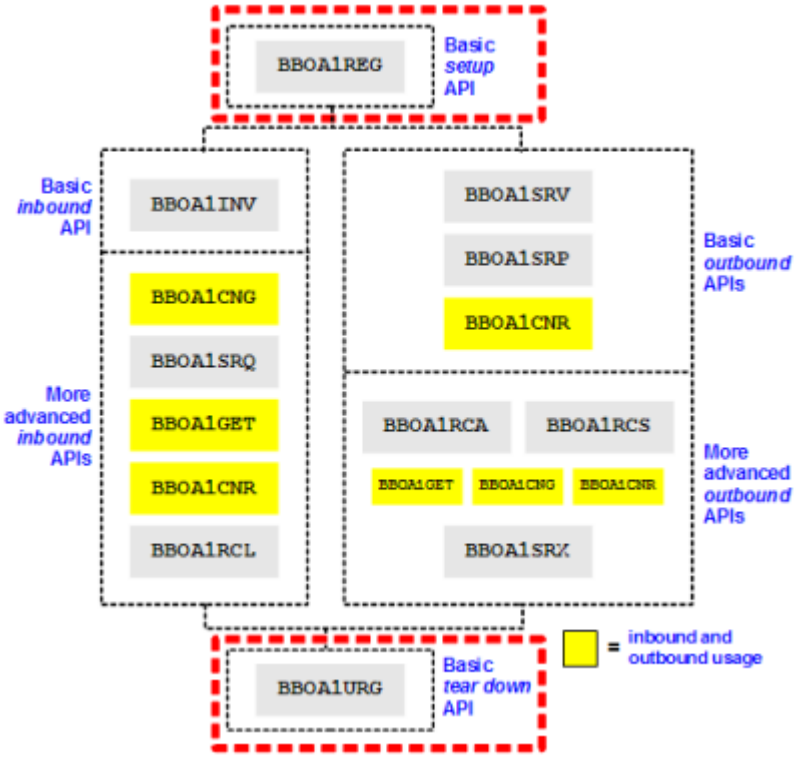
14 InfoCenter: `cdat_olaapis`. Each API has a table with RC and RSN. Very good explanations.

15 InfoCenter: `tdat_enableconnector`. The environment variable is needed to tell the Daemon to allow the use of WOLA. Daemon restart needed after setting this variable.

16 It is possible to have multiple registrations into the same daemon group. You just can't have two with the *same name* in the same daemon group.

Summary of Unit 1 - `BBOA1REG` and `BBOA1URG` APIs

In this unit we explored two of the 13 APIs:



We didn't have you do any "real work" with the other APIs between the REG and URG calls.

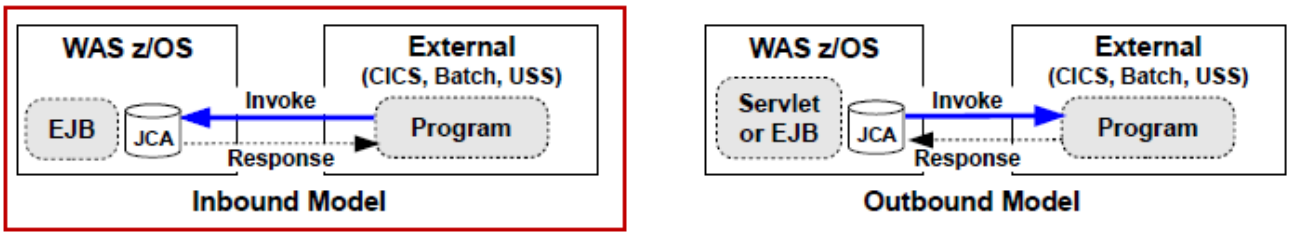
Key points we were trying to reinforce:

- Registration and unregistration mark the beginning and end of WOLA "session" with the WAS server environment.
- Registration is very specific to the cell, node *and server*. Your external program may only interact with a Java program in the server named on registration[17].
- The APIs have input and output parameters, and the InfoCenter spells out the data requirements and usage for each.
- `BBOA1REG` and `BBOA1URG` throw RC/RSN values. The InfoCenter has an excellent description of what each means.
- And (we hope) you saw that using these two APIs wasn't difficult at all.

17 You may have that Java program interact with *other* Java programs elsewhere in the WAS environment. That's WAS business as usual. The point here is that *WOLA* is very specific -- external address space to application server.

Unit 2: Exploring the *inbound* API model

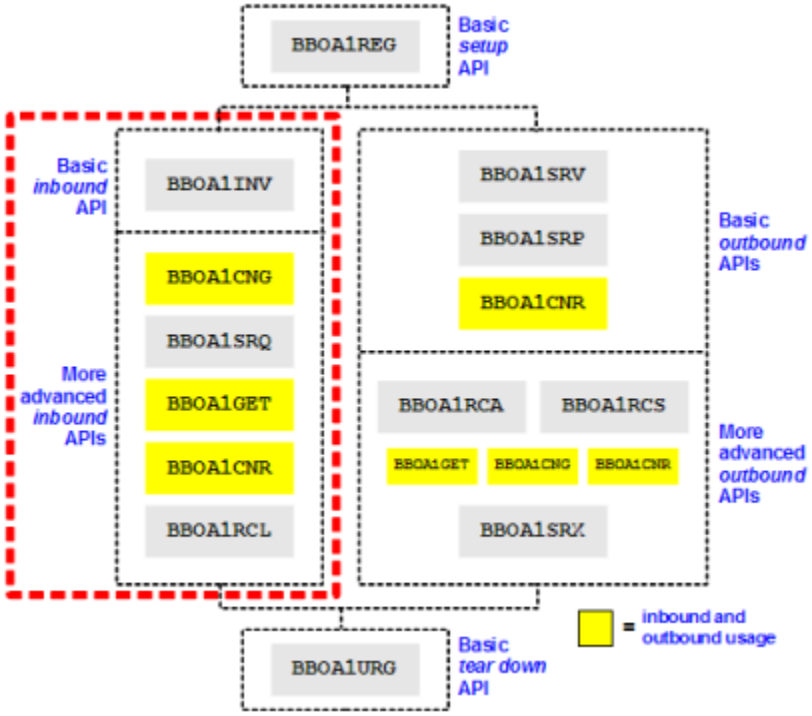
As a refresher, the inbound model is this:



In practical terms that means:

- The external program needs to **register** into the WAS cell, naming the cell short, node short and server short names. We saw how to do that in Unit 1 of this document.
- The external program needs to use that registration to send a **request** to the WAS side.
- The external program needs to process the returned **response**
- When all the work is done, the external program needs to **unregister**

The APIs we'll explore will be these:



We'll start with `BBOA1INV` because that's the easiest to understand and use.

Preparing the environment for exercises

To call from COBOL into WAS we need an application in the target server that's capable of dealing with the inbound WOLA call, and will do something we expect in return.

WOLA ships with a sample application called OLASample1.ear[18]. It's a handy tool that interacts with the supplied samples for both inbound and outbound work. That will be the Java program we'll work with throughout this document.

18 Found in the `/<was_smpe_root>/mso/OLA/samples/` directory

Do the following:

- ☐ Locate the `OLASample1.ear` file in the directory we indicated on the previous page.
- ☐ Install it into the application server you intend to use as the target for your WOLA calls. Take all the application defaults.
- ☐ Insure the application is started.

Overview of Exercise 2a - Simple BBOA1REG, BBOA1INV, BBOA1URG

The use of `BBOA1REG` and `BBOA1URG` is the same as in Exercise 1a. Now we'll place a **`BBOA1INV`** between the two that will send a message to the OLASample1 program in Java and receive the message back.

The InfoCenter indicates this API has a few parameters different from what we saw on REG and URG:

Table 23. BBOA1INV API syntax. The syntax is explained in the Parameters section.

API	Syntax
BBOA1INV	`BBOA1INV ( registername, requesttype, requestservicename, requestservicenamel, requestdata, requestdatalen, responsedata, responsedatalen, waittime, rc, rsn, rv )`

We'll explain what those parameters do in a moment. But first we have to do some additional things to the COBOL program to recognize and use those parameters and that API.

Note: The source for this exercise and all others can be found in the ZIP file that accompanies this document under the WP101490 Techdoc at `ibm.com/support/techdocs`.

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER2A.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      * REG and URG parms
       01 daemongroup                 PIC X(8) VALUE LOW-VALUES.
       01 node-name                   PIC X(8).
       01 server-name                 PIC X(8).
       01 register-name               PIC X(12) VALUE SPACES.
  1    01 minconn                     PIC 9(8) COMP VALUE 1.
       01 maxconn                     PIC 9(8) COMP VALUE 10.
       01 regopts                     PIC 9(8) COMP VALUE 0.
       01 urgopts                     PIC 9(8) COMP VALUE 0.
       01 rc                          PIC 9(8) COMP VALUE 0.
       01 rsn                         PIC 9(8) COMP VALUE 0.
       01 rv                          PIC 9(8) COMP VALUE 0.
      * INV parms
       01 service-name                PIC X(255).
       01 service-name-length         PIC 9(8) COMP.
       01 rqst-area                   PIC X(100) VALUE SPACES.
       01 rqst-area-addr              USAGE POINTER.
  2    01 rqst-area-length            PIC 9(8) COMP VALUE 100.
       01 resp-area                   PIC X(100) VALUE SPACES.
       01 resp-area-addr              USAGE POINTER.
       01 resp-area-length            PIC 9(8) COMP VALUE 100.
       01 wait-time                   PIC 9(8) USAGE BINARY.
       01 rqst-type                   PIC 9(8) COMP VALUE 1.
      * Working Variables
  3    01 text-msg                    PIC X(40) VALUE SPACES.
```

```
* Procedures Section
 PROCEDURE DIVISION.
 MAINLINE SECTION.
     MOVE 'EXER2A'                       TO register-name.
     MOVE 'S1CELL'                       TO daemongroup.
     MOVE 'S1NODEC'                      TO node-name.
     MOVE 'S1SR01C'                      TO server-name.
4    MOVE 'This is a test message'       TO text-msg.
     MOVE 'ejb/com/ibm/ola/olasample1_echoHome'
          TO service-name.
```

(BBOA1REG code same as before ... clipped to save space in document)

```
     MOVE text-msg TO rqst-area.
     MOVE LENGTH OF rqst-area TO rqst-area-length.
5    MOVE rqst-area-length TO resp-area-length.
     SET rqst-area-addr TO ADDRESS OF rqst-area.
     SET resp-area-addr TO ADDRESS OF resp-area.
     INSPECT service-name CONVERTING ' ' to LOW-VALUES.

     CALL 'BBOA1INV' USING
         register-name,
         rqst-type,
         service-name,
         service-name-length,
6        rqst-area-addr,
         rqst-area-length,
         resp-area-addr,
         resp-area-length,
         wait-time,
         rc,
         rsn,
         rv.

     IF rc > 0 THEN
       DISPLAY "OLA - BBOA1INV problem, rc/rsn/rv: " rc "/" rsn
       GO TO Bad-RC
     ELSE
7      DISPLAY "Message sent: " rqst-area
       DISPLAY "Message back: " resp-area
     END-IF.
```

(BBOA1URG code same as before ... clipped to save space in document)

Notes:

1. The working storage definitions for REG and URG are the same as before.
2. These are now required because of the parameters used on the `BBOA1INV` API. They are:

`register-name`	This must be equal to what was used on `BBOA1REG`
`rqst-type`	At present always "1" for EJB
`service-name`	The JNDI name of the deployed target EJB (case matters)
`service-name-length`	The length of the EJB's JNDI name
`rqst-area-addr`	Pointer to where the inbound message is held in memory
`rqst-area-length`	Length of the inbound message
`resp-area-addr`	Pointer to where the response message will go
`resp-area-length`	Length of response message expected
`wait-time`	The number of seconds to wait before timing out
`rc, rsn, rv`	Return code, reason code, and length of response

> **Note:** `BBOA1INV` makes a few assumptions, which is why the API is easy to use. In a moment we'll explain what those assumptions are. That will lead into a discussion of the more "advanced" inbound APIs.

3. Here we're simply setting up an area where a string literal will go.

4. The BBOA1REG variable values are as before. Here we're populating two more: text-msg is just that, a message we'll send over to WAS; service-name is the JNDI name of the EJB deployed in the target server. That's how WAS knows *which* EJB to invoke when the request comes in.
5. We're doing some housekeeping here, which takes a bit of explaining:

   MOVE text-msg TO rqst-area.
   Just a way to keep the message text separate from the parameter variable until the moment we're about to invoke. Strictly speaking not necessary ... we could have passed text-msg as parameter variable.

   MOVE LENGTH OF rqst-area TO rqst-area-length.
   This is how the length of the message is computed. The value is placed in the parameter variable.

   MOVE rqst-area-length TO resp-area-length.
   We know the OLASample1 application is a simple echo, so it was safe for us to assume input=output length. If you know a maximum length you set response length to that. If you're just not sure you'll have to use the "advanced" APIs, which return the length of the response.

   SET rqst-area-addr TO ADDRESS OF rqst-area.
   Computing the pointer to the input text area.

   SET resp-area-addr TO ADDRESS OF resp-area.
   Computing the pointer to the output text area.

   INSPECT service-name CONVERTING ' ' to LOW-VALUES.
   Just a bit of trickery ... if the service name is null-terminated then it's not necessary to compute and pass the length of the service name.
6. The BBOA1INV API is called with its parameters passed in.
7. Another simple IF, with the inbound text and outbound text displayed.

The output in the EXER2A batch program SYSOUT should looks like this:

```
Successfully registered into S1CELL
Message sent: This is a test message
Message back: This is a test message
Successfully unregistered from S1CELL
```

The output in the target WAS server's servant region will look like this:

```
Returning passed data:        ¢@   ¢@  @     ¢  @     ¢ ¢      @@@@@@@@@@
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@...
```

That's because the batch program is working in EBCDIC and the WAS z/OS JVM is in ASCII. The EBCDIC characters coming over are *not* translated to ASCII automatically[19].

Perform Exercise 2a

Do the following:

- ☐ Copy the supplied sample exer2a.txt to your system. Update the compile and link edit values according to your system.
- ☐ Update the daemongroup, node-name and server-name values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the runprog.txt JCL supplied in the Techdoc ZIP file.
- ☐ If you received anything other than RC=0, then consult the InfoCenter's API page and analyze the cause of the problem. Correct, recompile and re-run.

19 We could have illustrated codepage conversion before sending to WAS. But the objective of the inbound exercises is to explore the API usage and syntax. So we chose not to worry about the funny characters in the servant output. In the outbound exercises we'll see the COBOL doing code page conversions.

The assumptions `BBOA1INV` makes

We mentioned that `BBOA1INV` makes some assumptions. That's what makes it a simple API to use. Here are the assumptions:

- The response length is predictable; or at least the *maximum* response length is predictable.
- You wish the invoke to work *synchronously*; that is, your program invokes `BBOA1INV` and control does not come back until WOLA indicates the response is received.

If either of those doesn't hold true, then you must start considering the "advanced" APIs we'll discuss in a bit. Those allow you to operate with uncertain response lengths, and operate asynchronously ... meaning, you send the message and control is returned immediately to you. That frees you to go do other things. But it means you must come back and check to see if a response has been received. We'll illustrate all that later in the document.

Wrap-up of Exercise 2a

The `BBOA1INV` API does things for you under the covers. In fact, it uses the "advanced[20]" APIs to achieve the simplified appearance that `BBOA1INV` provides.

Here's a high-level summary of the things that take place behind the scenes with `BBOA1INV`:

- At time of registration a pool of connections is established. `BBOA1INV` retrieves one of those connections from the pool.
- The request is sent with the "asynch = no" flag set. That means your program does not gain control until WOLA signals a response is ready to process.
- A "get" is issued to pull the response from WOLA and put it into your working storage.

All those functions can be done by you with the "advanced" APIs, which we'll see in a bit.

`BBOA1INV` does it all for you, which is why it's such a simple API to use.

Exercise 2a illustrated a single "invoke" ... useful for validating the environment but not representing reality. WOLA shines when it gets to work again and again and again across the established registration.

That's what we'll do next. We're going to put in a loop.

Overview of Exercise 2b - Looping `BBOA1INV`

It's important to understand that `BBOA1INV` may be issued again and again within an established registration.

Exercise 2b is going to put a loop structure around the `BBOA1INV` block of code.

`BBOA1REG` and `BBOA1URG` portions of code

The `BBOA1REG` and `BBOA1URG` portions of the code remain unchanged from earlier exercises. The working storage definitions for those APIs is exactly the same.

`BBOA1INV` portion of code

The `BBOA1INV` portion of code remains largely unchanged.

We modified the `text-msg` a bit to include the loop count number into the message itself. This allows us to see that `BBOA1INV` is indeed processing unique requests each time.

20 The InfoCenter refers to them as "primitives." Main point: APIs with more granular control of behavior.

(portions of unchanged code clipped to save space)

```
* Working Variables
 01 text-msg                        PIC X(40) VALUE SPACES.
 01 text-msg-with-counter           PIC X(45) VALUE SPACES.
 01 stop-loop-flag                  PIC 9(1) COMP VALUE 0.
 01 loop-limit                      PIC 9(4) COMP VALUE 0.
 01 loop-counter                    PIC 9(4) COMP VALUE 0.
 01 loop-counter-text               PIC X(5) VALUE SPACES.
```

1

(portions of unchanged code clipped to save space)

2

```
         MOVE 10                    TO loop-limit.
```

(portions of unchanged code clipped to save space)

3

```
        PERFORM UNTIL loop-counter EQUAL loop-limit

          COMPUTE loop-counter = loop-counter + 1

          MOVE loop-counter TO loop-counter-text

          STRING
            loop-counter-text DELIMITED BY SIZE
            text-msg DELIMITED BY SIZE
            INTO text-msg-with-counter
          END-STRING

          MOVE text-msg-with-counter TO rqst-area
          MOVE LENGTH OF rqst-area TO rqst-area-length
          MOVE rqst-area-length TO resp-area-length
          INSPECT service-name CONVERTING ' ' to LOW-VALUES

          SET rqst-area-addr TO ADDRESS OF rqst-area
          SET resp-area-addr TO ADDRESS OF resp-area

          CALL 'BBOA1INV' USING
              register-name,
              rqst-type,
              service-name,
              service-name-length,
              rqst-area-addr,
              rqst-area-length,
              resp-area-addr,
              resp-area-length,
              wait-time,
              rc,
              rsn,
              rv

          IF rc > 0 THEN
            DISPLAY "OLA - BBOA1INV problem, rc/rsn: " rc "/" rsn
            GO TO Bad-RC
          ELSE
            DISPLAY "Message sent: " rqst-area
            DISPLAY "Message back: " resp-area
          END-IF

        END-PERFORM.
```

5

6

Notes:

1. These are variables to control the loop and to get the loop[21] number into the `text-msg` string literal.
2. Looping for this run limited to 10 iterations.

21 There may well be a more elegant way in COBOL to loop. The author's last formal programming education consists of FORTRAN on punched cards back in 1979. Ahh ... the memories. ☺

3. The loop is contained with a `PERFORM` / `END-PERFORM` block.
4. Some minor housekeeping to prepare for the concatenation of the loop counter to the string literal.
5. The same as Exercise 2a, except the request message is the string with the loop counter.
6. `DISPLAY` the request in and response back.

The output in `SYSOUT` should looks like this:

```
Successfully registered into S1CELL
Message sent: 0001 This is a test message
Message back: 0001 This is a test message
Message sent: 0002 This is a test message
Message back: 0002 This is a test message
Message sent: 0003 This is a test message
Message back: 0003 This is a test message
Message sent: 0004 This is a test message
Message back: 0004 This is a test message
Message sent: 0005 This is a test message
Message back: 0005 This is a test message
Message sent: 0006 This is a test message
Message back: 0006 This is a test message
Message sent: 0007 This is a test message
Message back: 0007 This is a test message
Message sent: 0008 This is a test message
Message back: 0008 This is a test message
Message sent: 0009 This is a test message
Message back: 0009 This is a test message
Message sent: 0010 This is a test message
Message back: 0010 This is a test message
Successfully unregistered from S1CELL
```

Perform Exercise 2b

Do the following:

- ☐ Copy the supplied sample `exer2b.txt` to your system. Update the compile and link edit values according to your system.
- ☐ Update the `daemongroup`, `node-name` and `server-name` values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the `runprog.txt` JCL supplied in the Techdoc ZIP file.
- ☐ If you received anything other than RC=0, then consult the InfoCenter's API page and analyze the cause of the problem. Correct, recompile and re-run.
- ☐ Increase the value of loop-limit and re-run.

Wrap-up of Exercise 2b

Exercise 2b was essentially the same as 2a except for a bit more COBOL to build and control a loop. You saw that multiple `BBOA1INV` calls can be performed within a single REG / URG.

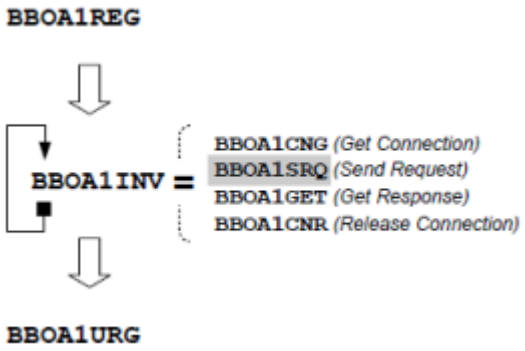
Moving on -- the "advanced" inbound APIs

As we mentioned, `BBOA1INV` really a packaging of other APIs into a simplified format.

Note: `BBOA1INV` may work perfectly well for what you need to accomplish. If so, then you don't really need to concern yourself with the more primitive inbound APIs.

But if you want finer control of the inbound behavior then the other inbound APIs come into play.

Here's a picture[22] that shows the relationship of BBOA1INV to these other APIs:



A brief explanation of those APIs ... (and remember, these too operate within a registration, so BBOA1REG and BBOA1URG is required, just as it was for BBOA1INV):

BBOA1CNG	"CNG" is short for "connection get." This requests a connection from the connection pool established with BBOA1REG. WOLA returns a connection to your program *with a connection handle*. That connection handle is used with the other APIs as you'll see.
BBOA1SRQ	"SRQ" is short for "send request." This sends the message to the target service over the connection named with connection-handle. BBOA1SRQ has two modes: *synchronous* and *asynchronous*. They determine when control is returned to your program. The simpler mode is *synchronous*, which means your program waits for WOLA to return control the response is ready. We'll see asynchronous used a bit later.
BBOA1GET	"GET" is short for ... well, "get." It pulls the response from the connection and places it in your working storage.
BBOA1CNR	"CNR" is short for "connection release." This returns the connection back to the pool.

Three quick notes:

- As you would imagine, using these APIs implies more parameters to work with.
- Your program assumes the responsibility of good housekeeping. For instance, failure to return connections to the pool could mean exhaustion of the thread pool.
- That said, it is possible to re-use the same connection over and over before releasing it back to the pool. That would be a way to increase efficiency even further.

Overview of Exercise 2c - single synchronous BBOA1SRQ and BBOA1GET

In this exercise we'll keep BBOA1REG and BBOA1URG just as they were.

We'll drop the use of BBOA1INV.

Instead, we'll use BBOA1CNG, BBOA1SRQ, BBOA1GET and BBOA1CNR for a single invocation[23] inbound to the target EJB in WAS.

A review of the Exercise 2c code:

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER2C.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      * API Parms
```

22 This is from the "Design and Planning Guide" PDF from the WP101490 Techdoc.
23 In Exercise 2d we'll wrap a loop around that.

```
(portions of unchanged code clipped to save space)
 01 connect-handle                PIC X(12) VALUE LOW-VALUES.
(portions of unchanged code clipped to save space)
 01 SRQasync                      PIC 9(8) COMP VALUE 0.
(portions of unchanged code clipped to save space)
*  BBOA1CNG - get connection

      MOVE 0 TO SRQasync.

      CALL 'BBOA1CNG' USING
            register-name,
            connect-handle,
            wait-time,
            rc,
            rsn.

      IF rc > 0 THEN
        DISPLAY "OLA - BBOA1CNG problem, rc/rsn: " rc "/" rsn
        GO TO Bad-RC
      END-IF.

*  BBOA1SRQ - setup and send

      INSPECT service-name CONVERTING ' ' to LOW-VALUES

      MOVE text-msg TO rqst-area
      MOVE LENGTH OF rqst-area TO rqst-area-length
      MOVE rqst-area-length TO resp-area-length

      SET rqst-area-addr TO ADDRESS OF rqst-area

      CALL 'BBOA1SRQ' USING
            connect-handle,
            rqst-type,
            service-name,
            service-name-length,
            rqst-area-addr,
            rqst-area-length,
            SRQasync,
            resp-area-length,
            rc,
            rsn

      IF rc > 0 THEN
        DISPLAY "OLA - BBOA1SRQ problem, rc/rsn: " rc "/" rsn
        GO TO Bad-RC
      END-IF

      SET resp-area-addr TO ADDRESS OF resp-area

      CALL 'BBOA1GET' USING
            connect-handle,
            resp-area-addr,
            resp-area-length,
            rc,
            rsn,
            rv

      IF rc > 0 THEN
        DISPLAY "OLA - BBOA1GET problem, rc/rsn: " rc "/" rsn
        GO TO Bad-RC
      ELSE
        DISPLAY "CNG,SRQ,GET sent: " rqst-area
        DISPLAY "CNG,SRQ,GET back: " resp-area
      END-IF
```

1 2 3 4 5 6 7

8
```
CALL 'BBOA1CNR' USING
      connect-handle,
      rc,
      rsn.

IF rc > 0 THEN
  DISPLAY "OLA - BBOA1CNR problem, rc/rsn: " rc "/" rsn
  GO TO Bad-RC
END-IF.
```
(BBOA1URG code the same as before)

Notes:

1. The `BBOA1CNG` API (get connection) returns the handle of the connection retrieved from the pool. This variable stores that value so the specific connection can be referenced on the other APIs.

   Further, the `BBOA1SRQ` API (send request) has two modes: *synchronous* and *asynchronous*[24]. A binary flag is used to determine which mode it will operate.
2. Here we set the async flag to `0`, which means *not* asynchonous, therefore it means run in *synchronous* mode. (Program issues SRQ and waits for control to be returned.)
3. We call the `BBOA1CNG` API to get a connection from the pool. It returns `connection-handle`.
4. Same kind of housekeeping we did for `BBOA1INV` in the earlier exercises.
5. We call the `BBOA1SRQ` API to send the request. We specify the connection-handle we got from CNG, and we specify *synchronous* mode by passing `0` for the async flag.
6. *Because SRQ was called synchronously*, we may assume control does not come back to us until a message is ready to get. So we may at this point call `BBOA1GET` with the connection-handle.
7. If RC=0 from `BBOA1GET` then we display a success message.
8. We call `BBOA1CNR` to release the connection back to the pool.

The output in `SYSOUT` should looks like this:

```
Successfully registered into S1CELL
CNG,SRQ,GET sent: This is a test message
CNG,SRQ,GET back: This is a test message
Successfully uregistered from S1CELL
```

Perform Exercise 2c

Do the following:

- ☐ Copy the supplied sample `exer2c.txt` to your system. Update the compile and link edit values according to your system.
- ☐ Update the `daemongroup`, `node-name` and `server-name` values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the `runprog.txt` JCL supplied in the Techdoc ZIP file.
- ☐ If you received anything other than RC=0, then consult the InfoCenter's API page and analyze the cause of the problem. Correct, recompile and re-run.

24 As mentioned, *synchronous* means control is held by WOLA until a response is received; *asynchronous* means control is returned immediately. Synchronous is "easier" because you simply issue and wait. Asynchronous requires a bit more programming on your part, but it allows your program to go "do other stuff" while the request is being processed by WAS.

Wrap-up of Exercise 2c

We had this coded in a way that used more APIs to do the same thing as `BBOA1INV`. In particular, the use of async=0 made this exercise just like the `BBOA1INV` exercises.

With the primitives and when operating in async=1 mode (we'll see this later) it's possible to multi-thread and have many requests into WAS at the same time while your program maintains control and does other work simultaneously. Obviously that requires more programming by you; specifically, it requires that your program come back and use the `BBOA1RCL` API to see if a response has returned for a specific connection handle (again, we'll see this later).

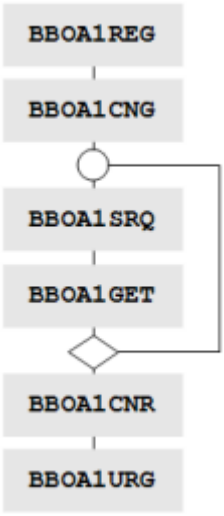
Overview of Exercise 2d -- looping BBOA1SRQ and BBOA1GET

We're now going to wrap a loop around these primitives. Just like with the `BBOA1INV` loop exercise, the API parameters don't change just because there's a loop. The same holds true here. So we won't review the code ... it's just like Exercise 2c but with a `PERFORM` loop.

But a question arises: where does the loop start and end? Do we include the connection get and connection release (CNG and CNR) *inside* or *outside* the loop?

It will work either way. But for maximum efficiency you would re-use the connection multiple times before returning it to the pool[25].

With this sample we'll get a single connection, loop, and then release the connection:



Perform Exercise 2d

Do the following:

- ☐ Copy the supplied sample `exer2d.txt` to your system. Update the compile and link edit values according to your system.
- ☐ Update the `daemongroup`, `node-name` and `server-name` values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the `runprog.txt` JCL supplied in the Techdoc ZIP file.
- ☐ If you received anything other than RC=0, then consult the InfoCenter's API page and analyze the cause of the problem. Correct, recompile and re-run.

25 Be aware that if you have multiple connections active concurrently then the connection-handle you use on a given `BBOA1GET` becomes very important. Your program must track and use the handles correctly.

Wrap-up of Exercise 2d

The key point emphasized with this exercise is that repeated invocations with `BBOA1SRQ` and `BBOA1GET` works. We also drew your attention to the placement of CNG and CNR with respect to loops.

Exercise 2c and 2d were both *synchronous* invocations of `BBOA1SRQ`. Let's now explore asynchronous `BBOA1SRQ`.

Overview of "synchronous" and "asynchronous" with respect to `BBOA1SRQ` and `BBOA1RCL`

The fundamental issue here is when control is returned to your program:

- Asynchronous -- right away
- Synchronous -- when a response is ready to be processed

Synchronous is the "simplest" in that you call the API and do nothing until control is returned back to you. Then you may assume there's a response available and process that.

Asynchronous implies calling the API and having control returned back to your program *immediately*. That frees your program to go do other things without having to wait on each response.

Note: This allows you to issue multiple requests over different connection handles and then come back and process the responses when they come back. It changes your program from being sequentially dependent.

However, it puts the responsibility on *your program* to check back to see if the response has returned. That is done with the `BBOA1RCL` API. That API is a check for the length of a response message.

Here's what the InfoCenter has for `BBOA1RCL`:

Table 19. BBOA1RCL API syntax. The syntax is explained in the Parameters section.

API	Syntax
BBOA1RCL	`BBOA1RCL ( connectionhandle, async(0|1), responsedatalen, rc, rsn )`

It checks the named `connectionhandle` to see if a response is waiting and if so it returns the length with the `responsedatalen` output parameter.

But if no response has yet arrived, the `responsedatalen` output value is set to all `x'FF'`.

But there's a twist ... `BBOA1RCL` has an `async` parameter just like `BBOA1SRQ` did. There is a hierarchical relationship between the two APIs and their `async` parameters. The following table illustrates that relationship.

<table>
<tr><td rowspan="3">BBOA1SRQ,async=0
(synchronous)
Control is held until response received.
When response received the response length is returned on <code>responsedatalen</code> output parameter of <code>BBOA1SRQ</code>.

BBOA1RCL is not necessary.
Control is held by WOLA until response is received.
Response length is returned on <code>responsedatalen</code> output parameter of <code>BBOA1SRQ</code>.</td><td colspan="3">BBOA1SRQ,async=1
(asynchronous)
Control is returned to your program immediately. For responses that require time to generate and return, this allows your program to process other work during wait for response to come back.
But ... it is your responsibility to determine when response is ready.
Therefore you must use <code>BBOA1RCL</code> for that purpose
Two modes of BBOA1RCL usage ...</td></tr>
<tr><td colspan="2">BBOA1RCL,async=0
(synchronous)</td><td>BBOA1RCL,async=1
(asynchronous)</td></tr>
<tr><td>If response available then length provided on <code>responsedatalen</code> output parameter of <code>BBOA1RCL</code></td><td>If response not available then control is held until response is available. At that time the length provided on <code>responsedatalen</code> output parameter of <code>BBOA1RCL</code></td><td>If response available then length provided on <code>responsedatalen</code> output parameter of <code>BBOA1RCL</code></td><td>If response not available then <code>responsedatalen</code> output parameter of <code>BBOA1RCL</code> set to all <code>x'FF'</code> and control returns to your program.
You must come back and call <code>BBOA1RCL</code> again and repeat until message length returned.</td></tr>
<tr><td>We saw this with
Exercises 2c and 2d
(single invoke and looping)</td><td colspan="2">We'll see this with
Exercises 2e and 2f
(single invoke and looping)</td><td>We'll see this with
Exercises 2g and 2h
(single invoke and looping)</td></tr>
</table>

Overview of Exercises 2e and 2f - asynchronous BBOA1SRQ with synchronous BBOA1RCL

This sample will show the issuance of `BBOA1SRQ` with `async=1` set, then followed up with `BBOA1RCL` with `async=0`. This is the simpler of the two `BBOA1RCL` scenarios because we only need to worry about issuing `BBOA1RCL` once: with `async=0` control is held until a message is returned.

Note: With `async=1` on `BBOA1RCL` the message length may come back `x'FFFFFFFF'`, indicating no message is yet ready. So you have to repeat the `BBOA1RCL` until you get a message length indicating the response is ready.

A review of the Exercise 2e code:

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER2E.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      * API Parms
           : (other definitions same as before)
       01 SRQasync                      PIC 9(8) COMP VALUE 0.
1      01 RCLasync                      PIC 9(8) COMP VALUE 0.
           : (other definitions same as before)
      * Procedures Section
       PROCEDURE DIVISION.
       MAINLINE SECTION.
           : (BBOA1REG same as before)
      *  Asynch BBOA1SRQ with synchronous BBOA1RCL

2          MOVE 1 TO SRQasync.
           MOVE 0 TO RCLasync.
```

```
           CALL 'BBOA1CNG' USING ...
              : (BBOA1CNG same as before)
              : (housekeeping to issue BBOA1SRQ same as before)

3          CALL 'BBOA1SRQ' USING ...
              : (BBOA1SRQ same as before, except variable SRQasync is now 1 rather than 0)

4          PERFORM Other-Work.
           DISPLAY "This is your program coming back from other work".

       * RCLasync set to 0 (synchronous) above

5          CALL 'BBOA1RCL' USING
                 connect-handle,
                 RCLasync,
                 resp-area-length,
                 rc,
                 rsn.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1RCL problem, rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           ELSE
             DISPLAY "Successfully back from synchronous BBOA1RCL"
           END-IF.

              : (what follows same as before -- GET, CNR, URG, etc.)

6      Other-Work.
           DISPLAY "Simulation of 'other work' when async=1".

/*
```

Notes:

1. The `SRQasync` variable we saw before; what's new is the `RCLasync`. We needed to have SRQ be 1 and RCL be 0 for this exercise. We have separate variables for each API simply to keep things clean and separate.
2. Here we set the SRQ value to 1 (asynchronous) and the RCL value to 0 (asynchronous).
3. `BBOA1SRQ` is called as before, but this time with `async=1`. That means program control is returned to us *immediately*. We are free to go off and do other work ...
4. ... which is what we do with a very simple `PERFORM` call that comes right back.
5. We then drop into `BBOA1RCL` to get the response length. The variable `RCLasync` was set to `0` earlier, so this call to RCL will be *synchronous*. That means program control does not return to us until a response has been received, and at that time we can pull the response length from output parameter `resp-area-length`. That is then used on the `BBOA1GET`, just as before.
6. Our funny little "other work" simulator.

The output in SYSOUT should looks like this:

```
Successfully registered into S1CELL
Successfully issued async BBOA1SRQ
Simulation of 'other work' when async=1
This is your program coming back from other work
Successfully back from synchronous BBOA1RCL
Message sent: This is a test message
Message back: This is a test message
Successfully unregistered from S1CELL
```

Perform Exercise 2e

Do the following:

- ☐ Copy the supplied sample `exer2e.txt` to your system. Update the compile and link edit values according to your system.
- ☐ Update the `daemongroup`, `node-name` and `server-name` values to match your WAS cell values.
- ☐ Compile the program. Fix any errors that result in a RC other than 0.
- ☐ Execute the program using the `runprog.txt` JCL supplied in the Techdoc ZIP file.
- ☐ If you received anything other than RC=0, then consult the InfoCenter's API page and analyze the cause of the problem. Correct, recompile and re-run.

Perform Exercise 2f

Exercise 2f is simply 2e except with a loop structure that starts after `BBOA1CNG` and ends before `BBOA1CNR`. It loops by default 10 times, and each time it branches off to do "other work" after issuing the asynchronous `BBOA1SRQ`.

- ☐ Copy, update, compile and execute as before.

Wrap-up of Exercises 2e and 2f

In these exercises we began the illustration of *asynchronous* operations. As noted earlier we have two APIs on which async is a parameter: `BBOA1SRQ` and `BBOA1RCL`. In Exercise 2e we set `async=1` on SRQ. That required us to then use `BBOA1RCL` to get the message length.

But we held `BBOA1RCL` to `async=0` to keep the exercise a bit simpler. Simpler because we could issue `BBOA1RCL`, `async=0` and simply wait for WOLA to tell us to proceed.

Exercise 2f simply put a loop structure around the SRQ-RCL-GET sequence.

Now we're ready to show SRQ and RCL *both* with `async=1`. That gives us maximum freedom to "go do other work" while WAS responds; but it also puts some responsibility on us to code checking logic into our program.

Overview of Exercise 2g - `BBOA1SRQ` and `BBOA1RCL` both set to `async=1`, no loop

These are nearly the same as 2e and 2f, except we'll set `RCLasync` to `1`. But it involves a bit more than that because we have to take into account the possibility that a response is not yet back from WAS. That implies a test of the `responsedataln` parameter of `BBOA1RCL`:

- If `responsedataln` equal to `x'FFFFFFFF'` then no response is yet returned. That means your program is free to go do other work. Eventually you have to come back and test again.
- If `responsedataln` is ***not equal*** to `x'FFFFFFFF'` then a response has been returned, and its length is specified by `responsedataln`. You then use `BBOA1GET` with the connection handle and response length to get the message.

Because this introduces a few new things we'll show the sample in its entirety.

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER2G.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
```

```
* API Parms
 01 daemongroup                  PIC X(8) VALUE LOW-VALUES.
 01 node-name                    PIC X(8).
 01 server-name                  PIC X(8).
 01 register-name                PIC X(12) VALUE SPACES.
 01 minconn                      PIC 9(8) COMP VALUE 1.
 01 maxconn                      PIC 9(8) COMP VALUE 10.
 01 regopts                      PIC 9(8) COMP VALUE 0.
 01 urgopts                      PIC 9(8) COMP VALUE 0.
 01 service-name                 PIC X(255).
 01 service-name-length          PIC 9(8) COMP.
 01 rqst-area                    PIC X(100) VALUE SPACES.
 01 rqst-area-addr               USAGE POINTER.
 01 rqst-area-length             PIC 9(8) COMP VALUE 100.
 01 resp-area                    PIC X(100) VALUE SPACES.
 01 resp-area-addr               USAGE POINTER.
 01 resp-area-length             PIC 9(8) COMP VALUE 100.
 01 resp-area-length-char REDEFINES resp-area-length PIC X(4).
 01 connect-handle               PIC X(12) VALUE LOW-VALUES.
 01 wait-time                    PIC 9(8) USAGE BINARY.
 01 rqst-type                    PIC 9(8) COMP VALUE 1.
 01 SRQasync                     PIC 9(8) COMP VALUE 0.
 01 RCLasync                     PIC 9(8) COMP VALUE 0.
 01 rc                           PIC 9(8) COMP VALUE 0.
 01 rsn                          PIC 9(8) COMP VALUE 0.
 01 rv                           PIC 9(8) COMP VALUE 0.
* Working Variables
 01 text-msg                     PIC X(40) VALUE SPACES.
 01 good-RCL-flag                PIC 9(1) COMP VALUE 0.
 01 RCL-attempts                 PIC 9(4) COMP VALUE 0.
 01 other-work-counter           PIC 9(8) COMP VALUE 0.

* Procedures Section
 PROCEDURE DIVISION.
 MAINLINE SECTION.
     MOVE 'EXER2G'                       TO register-name.
     MOVE 'S1CELL'                       TO daemongroup.
     MOVE 'S1NODEC'                      TO node-name.
     MOVE 'S1SR01C'                      TO server-name.
*
     MOVE 'This is a test message'       TO text-msg.
     MOVE 'ejb/com/ibm/ola/olasample1_echoHome'
          TO service-name.

     INSPECT daemongroup CONVERTING ' ' to LOW-VALUES.

     CALL 'BBOA1REG' USING
           daemongroup,
           node-name,
           server-name,
           register-name,
           minconn,
           maxconn,
           regopts,
           rc,
           rsn.

     IF rc > 0 THEN
       DISPLAY "OLA - BBOA1REG problem -- rc/rsn : " rc "/" rsn
       GO TO Bad-RC
     ELSE
       DISPLAY "Successfully registered into " daemongroup
     END-IF.
```

1
2
3

```
*  Asynch BBOA1SRQ with synchronous BBOA1RCL

4       MOVE 1 TO SRQasync.
        MOVE 1 TO RCLasync.

        CALL 'BBOA1CNG' USING
              register-name,
              connect-handle,
              wait-time,
              rc,
              rsn.

        IF rc > 0 THEN
          DISPLAY "OLA - BBOA1CNG problem, rc/rsn: " rc "/" rsn
          GO TO Bad-RC
        END-IF.

        INSPECT service-name CONVERTING ' ' to LOW-VALUES.

        MOVE text-msg TO rqst-area.
        MOVE LENGTH OF rqst-area TO rqst-area-length.

        SET rqst-area-addr TO ADDRESS OF rqst-area.
        SET resp-area-addr TO ADDRESS OF resp-area.

        CALL 'BBOA1SRQ' USING
              connect-handle,
              rqst-type,
              service-name,
              service-name-length,
              rqst-area-addr,
              rqst-area-length,
              SRQasync,
              resp-area-length,
              rc,
              rsn.

        IF rc > 0 THEN
          DISPLAY "OLA - BBOA1SRQ problem, rc/rsn: " rc "/" rsn
          GO TO Bad-RC
        END-IF.

5       PERFORM UNTIL good-RCL-flag EQUAL 1

6         PERFORM Other-Work

          CALL 'BBOA1RCL' USING
                connect-handle,
7               RCLasync,
                resp-area-length,
                rc,
                rsn

          IF rc > 0 THEN
            DISPLAY "OLA - BBOA1RCL problem, rc/rsn: " rc "/" rsn
            GO TO Bad-RC
          END-IF

8         IF resp-area-length-char EQUAL HIGH-VALUES THEN
            COMPUTE RCL-attempts = RCL-attempts + 1
          ELSE
            DISPLAY "Good RCL on attempt: " RCL-attempts
9           MOVE 0 TO RCL-attempts
            MOVE 1 TO good-RCL-flag
          END-IF

10      END-PERFORM.
```

```
           CALL 'BBOA1GET' USING
                 connect-handle,
                 resp-area-addr,
                 resp-area-length,
                 rc,
                 rsn,
                 rv

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1GET problem, rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           ELSE
             DISPLAY "Message sent: " rqst-area
             DISPLAY "Message back: " resp-area
           END-IF

           CALL 'BBOA1CNR' USING
                 connect-handle,
                 rc,
                 rsn.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1CNR problem, rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           END-IF.

       *  Unregister from the Daemon group using BBOA1URG API           *

           CALL 'BBOA1URG' USING
               register-name,
               urgopts,
               rc,
               rsn.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1URG problem -- rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           ELSE
             DISPLAY "Successfully unregistered from " daemongroup
           END-IF.

           GOBACK.

       *  Used to doing other work when asych=1 specified

        Other-Work.
           PERFORM UNTIL other-work-counter EQUAL 10000
11           COMPUTE other-work-counter = other-work-counter + 1
           END-PERFORM.
           MOVE 0 TO other-work-counter.

       *  Section used to exit batch if any API returned RC>0

        Bad-RC.

           DISPLAY "OLA - EXITING program due to non-RC=0."
           GOBACK.

/*
```

Notes:

1. All the values defined at the top of the working storage section are as they have been in prior exercises.

2. The `resp-area-length` definition is the same as before, but we've added `resp-area-length-char` to provide us with a simple way to check for equality with `x'FFFFFFFF'`.
3. We added some variables to control and report on the `PERFORM` loop we used when checking if `BBOA1RCL` came back with a response length.
4. Both `SRQasync` and `RCLasync` are set to `1` for asynchronous operations.
5. This is the start of the `PERFORM` loop in which the `BBOA1RCL` check is done.
6. This `PERFORM` points down to block **11** which simulates "other work" being done while the asynchronous operations of `BBOA1SRQ` and `BBOA1RCL` are in process..

   Note: This is a very poor illustration, but it is adequate for the purposes of this document.
7. The `BBOA1RCL` API is called with the value of `RCLasync` set to `1` (asynchronous)
8. We check if `resp-area-length-char`[26] is equal to `HIGH-VALUES`. If there is *no response* then this test will be *true*. Therefore, if `resp-area-length-char = HIGH-VALUES` then *no response ready* so increment a counter by 1 and continue.
9. However, If it does *not* equal `HIGH-VALUES` then we assume a response *has* been received and the value of `resp-data-length` is used on the `BBOA1GET` API. The number of RCL attempts is reported, then we clear the `RCL-attempts` counter. Then we set the `good-RCL-flag` to `1`.
10. The `END-PERFORM` for the earlier `PERFORM` that did the `good-RCL-flag` check. If we received a good response length then the flag value will be 1 and we'll drop out of this `PERFORM`. But if it's `0` we go do "other work" and then come back to try again.
11. This is our simple "other work" ... a counter that increments up to `10000` then clears the counter and returns.

The output in SYSOUT may looks *something* like this:

```
Successfully registered into S1CELL
Good RCL on attempt: 0016 [27]
Message sent: This is a test message
Message back: This is a test message
Successfully unregistered from S1CELL
```

Overview of Exercise 2h - `BBOA1SRQ` and `BBOA1RCL` both set to `async=1`, with loop

This is Exercise 2g but with a loop wrappered around the SRQ-RCL-GET sequence.

The output looks a little different because of the loop:

```
Successfully registered into S1CELL
Good RCL on attempt: 0036 (On the first iteration of the loop it took 36 "other work" attempts before response ready)
Message sent: 0001 This is a test message
Message back: 0001 This is a test message
 : (Iterations 2 through 9 clipped to save space)
Good RCL on attempt: 0012 (On the 10th iteration of the loop it took only 12)
Message sent: 0010 This is a test message
Message back: 0010 This is a test message
Successfully unregistered from S1CELL
```

Perform Exercises 2g and 2h

As before ...

☐ Copy, update, compile and execute as before.

---

26 That is, the the `PIC X(4)` value that redefines the `PIC 9(8)` original `resp-area-length` value.

27 How many "RCL attempts" you see with *this sample* is a function of how quickly WAS responds. Our "other work" is a poor illustration because it takes a compiled COBOL program very little time to count to 10000.

Wrap-up of Exercises 2g and 2h

These exercises showed one of the more complex scenarios; that is, using `BBOA1SRQ` and `BBOA1RCL` in asynchronous mode. As a recap:

- `BBOA1SRQ` with `async=1` created the need to then use `BBOA1RCL`
- `BBOA1RCL` with `async=1` required the code to take into account that the response may not yet be back.

There was an additional level of level of complexity we could have shown but did not: issuing multiple `BBOA1SRQ` with `async=1` and thus having multiple requests in WAS at once. The use of the APIs is no different -- but the complexity of the COBOL programming goes up a notch or two and ... well ... the author just isn't that good a programmer ☺.

Wrap-up of the Unit 2 -- Inbound APIs

In this unit we explored the inbound APIs. And by "inbound" we mean that the *request invocation is initiated from the external address space and destined for an EJB in WAS z/OS.*[28]

We saw that the APIs can be very simple or a bit more complex, depending on the degree of control you desire:

- `BBOA1INV` -- simple, but it is a synchronous model and it requires that you know the maximum length of the response.
- `BBOA1SRQ` -- more control, but it requires the use of `BBOA1CNG`, `BBOA1GET`, `BBOA1CNR` and perhaps also `BBOA1RCL` depending on the factors we spelled out in the exercises.

We are now ready to begin exploring the *outbound* APIs. That means the initial request is initiated by the Java program in WAS z/OS, and the target is an external address space program such as batch, CICS or USS.

28 As mentioned earlier, the WAS EJB will return a response, so the WOLA channel goes both ways. The key for understanding "inbound" and "outbound" with respect to the APIs is ... *who initiates the request?*

Unit 3: Exploring the *outbound* API model

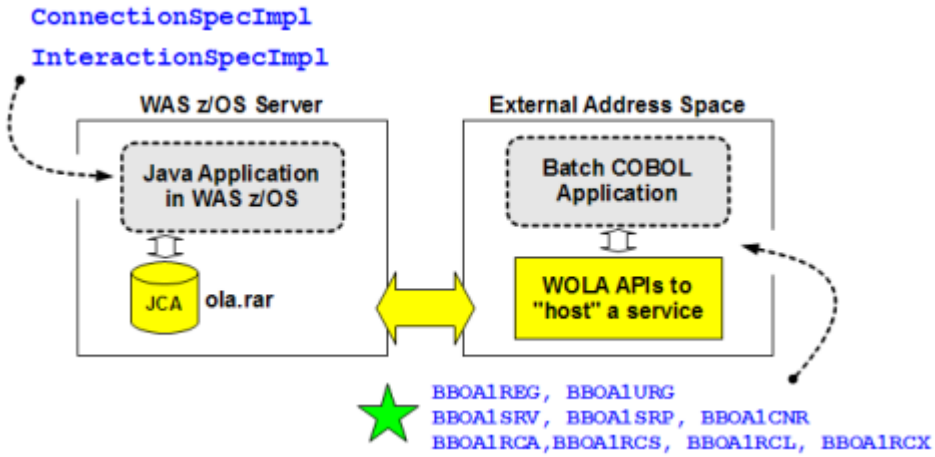
Right from the start we have to very clearly differentiate *which* APIs we're speaking of.

In Unit 2 we focused entirely on what we call the "native" APIs -- those used by COBOL[29]. We assumed the Java code was properly implemented. That was a safe assumption because we were using the supplied sample EJB application which echoed back what it was sent.

In this unit we'll turn our attention to the outbound APIs. As we noted earlier, "outbound" implies that the request is initiated from *inside WAS* and the target is the external address space. That implies that Java code will do the initiation.

The Java programming interfaces

There *are* Java classes and methods related to WOLA[30]. We'll cover some of that under "A picture representation of the relationships when using WOLA" on page 55. For now take a look at this picture:



On the Java side we have two key classes as shown. **`ConnectionSpecImpl`** is used to associate the Java program with the registration name ... that is, what was used by the other program on `BBOA1REG`. **`InteractionSpecImpl`** is what associates the Java program with the "service name" ... that is, what was used by the other program on the `BBOA1SRV`[31]. That's a very simplified explanation, but good enough for now.

What we're interested in are the APIs used on the *non-Java* side. Those are APIs used to "host a service" and send a response back.

Important: When the external address space is CICS

If the program you wish to communicate with outside WAS z/OS resides in CICS, you *may not have to write to the APIs at all*. That's because the WOLA support for CICS includes function that "hides" all that from you.

If your CICS program can be invoked with a CICS EXEC LINK, then it is eligible to be invoked over WOLA using the `BBO$` Link Server function. See "A picture representation of the relationships when using WOLA" on page 55 for more on this.

Preparing the environment

It's the same as we saw under "Exercise Preparation" on page 18.

29 Or C/C++, High Level Assembler or PL/I, but this document is focused on COBOL.
30 InfoCenter, search on `tdat_useoutboundconnection`
31 Or `BBOA1RCA`/`BBOA1RCS` as you'll see in this unit.

The concept of "hosting a service"

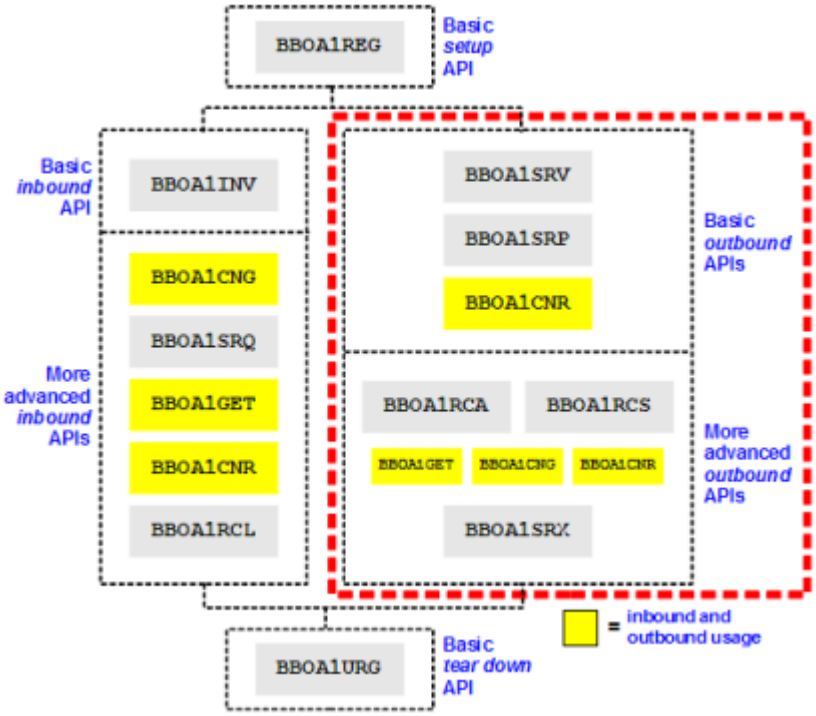
In Unit 2 the COBOL program sent requests into WAS and we simply assumed *something* was on the WAS side to catch the request and respond. That *something* was the WOLA JCA adapter and the `OLASample1.ear` sample application.

When we go the *other direction* -- WAS into batch COBOL -- *something* has to be there ready to catch the request and respond. That *something* is the batch COBOL program. And it involves using the *outbound* APIs[32].

At its most basic, this involves two things:

1. Having the COBOL program register into WAS. That's the `BBOA1REG` API, and its done in *exactly the same way we saw back in the earlier exercises*.
2. Then the COBOL program invokes an API to put itself into a state of readiness to receive a request. That's a kind of "wait until something arrives" state.

The APIs we'll look at in this unit are these:



Registering and Unregistering with `BBOA1REG` and `BBOA1URG`

It's exactly the same as we showed in Unit 2. If the environment is CICS then there is a potential slight variation to this[33].

The important thing to understand about this is that the Java program in WAS *must* know the registration name used in order to send a request over. This is drawn out in picture form under "A picture representation of the relationships when using WOLA" on page 55.

The "service name" role when going outbound

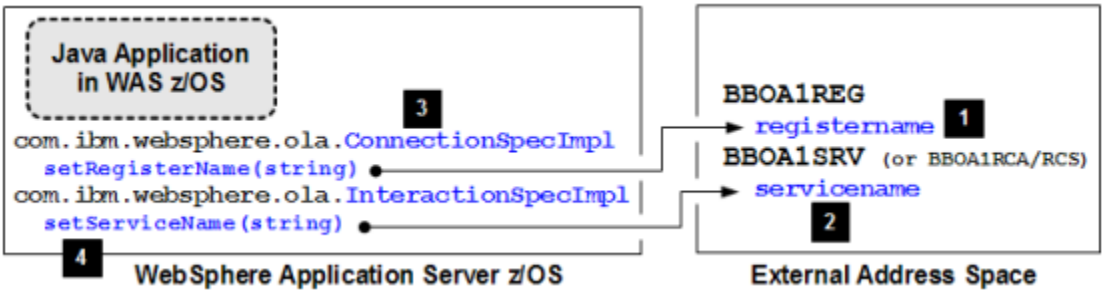
With the *inbound* model we saw that the "service name" we specified on the `BBOA1INV` API was the JNDI name of the target EJB. That's what allowed WOLA to know *which* Java target to invoke.

32 Reminder: our use of "inbound" and "outbound" is always from the perspective of WAS z/OS.

33 Having to do with using the `BBOC START_SRVR` command to perform the registration. The `BBOA1REG` API is still used, but it's used within the `BBOC` control transaction code and thus "under the covers."

With *outbound* we have the same issue -- the calling Java program needs to indicate what it is to call. Any given registration may have lots of services available. What constitutes the service name here?

It's a parameter on the `BBOA1SRV` API[34]. That creates a "service" within a registration space. Java programs in WAS therefore need two key pieces of information to invoke the program in COBOL -- the registration name used on the `BBOA1REG` API, and the service name used on the `BBO1ASRV` API. In picture form[35]:



Notes:

1. The external program uses `BBOA1REG` and provides a register name.
2. The external program "hosts a service" and provides a service name.
3. The Java program uses `ConnectionSpecImpl` and `setRegisterName()` to specify the registration pool to connect to.
4. The Java program uses `InteractionSpecImpl` and `setServiceName()` to specify the service within the registration pool it has connected to.

There's an interesting twist to this: if the `BBOA1SRV` API specifies an asterisk for `servicename` parameter, that means it'll accept a request on *any service for that registration*. The Java-side program is free to use any string on `setServiceName()`.

We have to be a little careful with that asterisk. Any hosted service using asterisk will grab any request it sees on that registration pool. If that's what you want, then use asterisk. But if you want a registration pool to service *separate* hosted services then that separation is created by using specific `servicename` values on the `BBOA1SRV` API.

Overview of Exercise 3a - `BBOA1SRV` and `BBOA1SRP`

This exercise will illustrate a single-pass use of `BBOA1SRV` and `BBOA1SRP`. In the next exercise we'll introduce a loop.

The `BBOA1SRV` parameters

First, let's take a quick look at what the InfoCenter has to say about the `BBOA1SRV` API:

Table 25. BBOA1SRV API syntax. The syntax is explained in the Parameters section.

API	Syntax 1 2
BBOA1SRV	BBOA1SRV (registername, requestservicename, requestservicenamel, requestdata, requestdatalen, connectionhandle, waittime, rc, rsn, rv) 3 4

Notes:

1. `BBOA1SRV` requires that a `BBOA1REG` was called and a registration pool established. This parameter tells `BBOA1SRV` which registration pool to use[36].

34 Or `BBOA1RCA` or `BBOA1RCL`, which are finer-controlled variations on `BBOA1SRV`.

35 See "A picture representation of the relationships when using WOLA" on page 55.

2. This is the "interesting twist" we mentioned earlier -- this may be a specific string *or* it may be an asterisk:
   - *Specific string* -- `BBOA1SRV` listens on just that service name, which means the Java program must use that exact service name to communicate with this hosted service.
   - *Asterisk* -- `BBOA1SRV` listens across all service names that come into the registration pool.

   For this exercise we'll do both.
3. The length of the data area to receive the message into. This implies you have a sense for the length of the data, or at least a sense for the maximum length.
4. The connection handle that is returned.

How `BBOA1SRV` works

Here's a quick couple of points about `BBOA1SRV` that will help you understand its use a bit better:

- It is by nature *synchronous*; that is, when called it takes program control away from your calling program until the Java side sends a request to it.
- It calls `BBOA1CNG` under the covers ... that's what gets the connection implied by the `connectionhandle` output parameter.
- It calls `BBOA1GET` under the covers ... that's what pulls the message in.
- It does not "answer back" ... it just processes the message outbound from WAS. We'll use `BBOA1SRP` to send an answer back.

The primitives (`BBOA1RCA` and `BBOA1RCS`) allow greater control over these.

The Java program used to drive this

This is an *outbound* model, which means we have to tell the Java program to initiate a request over to the COBOL program. Further, we have to tell the Java program which registration pool to use, what service name to use, and what message to send.

The `OLASample1.ear` sample program we've used up to this point has a web page to do this sort of thing. *But it assumes CICS with the `BBO$` link server task.* It uses some of the CICS-specific methods which throw errors if there's no `BBO$` link server task present.

So we're supplying a modified version of `OLASample1.ear` called **`ATSsample1.ear`** that will work with COBOL batch. We supply that in the ZIP file that accompanies the Techdoc.

> **Note:** The ZIP file that accompanies this document on Techdocs includes two sample programs -- `ATSsample1.ear`, and `ATSSample1-new.ear`. The difference is a single TransactionAttribute of `NOT_SUPPORTED`. In the README for the ZIP we say the "new" copy is for V8 while the earlier copy was for V7, but you may find it necessary to use the "new" copy in later maintenance levels of V7.

In the "Perform Exercise 3a" section we'll show how to access the web page and use it.

ASCII / EBCDIC Conversion

COBOL is operating in EBCDIC and the WAS z/OS Java environment runs in ASCII. For the inbound we didn't bother doing any conversion. The EBCDIC characters displayed in the WAS server `SYSPRINT` in a funny way. But our objective of showing WOLA work was served well enough.

For the outbound we'll do some conversion so the message returned to the browser is readable. It's a way of showing things in a cleaner, more comfortable way.

---

36 Up to this point we've shown the COBOL programs issuing one `BBOA1REG` and establishing a single named registration pool. But in truth a program could establish multiple registrations with different names. Given that, we have to tell `BBOA1SRV` which one of what may be several registrations to listen on.

A review of Exercise 3a code

```
//COBOL.SYSIN  DD  *
       IDENTIFICATION DIVISION.
       PROGRAM-ID. EXER3A.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
      * REG and URG parms
       01 daemongroup                 PIC X(8) VALUE LOW-VALUES.
1      01 node-name                   PIC X(8).
       01 server-name                 PIC X(8).
       01 register-name               PIC X(12) VALUE SPACES.
       01 minconn                     PIC 9(8) COMP VALUE 1.
       01 maxconn                     PIC 9(8) COMP VALUE 10.
       01 regopts                     PIC 9(8) COMP VALUE 0.
       01 urgopts                     PIC 9(8) COMP VALUE 0.
       01 rc                          PIC 9(8) COMP VALUE 0.
       01 rsn                         PIC 9(8) COMP VALUE 0.
       01 rv                          PIC 9(8) COMP VALUE 0.
      * SRV and SRP parms
       01 SRV-service-name            PIC X(255).
       01 SRV-service-name-length     PIC 9(8) COMP.
2      01 SRV-rqst-area               PIC X(120).
       01 SRV-rqst-area-addr          USAGE POINTER.
       01 SRV-rqst-area-length        PIC 9(8) COMP.
       01 SRP-resp-area               PIC X(120).
       01 SRP-resp-area-addr          USAGE POINTER.
       01 SRP-resp-area-length        PIC 9(8) COMP.
       01 connect-handle              PIC X(12).
       01 wait-time                   PIC 9(8) USAGE BINARY.
      * ASCII/ECDIC Conversion Variables
       01 EBCDIC-CCSID                PIC 9(4) BINARY VALUE 1140.
       01 ASCII-CCSID                 PIC 9(4) BINARY VALUE 819.
3      01 INPUT-EBCDIC                PIC X(120).
       01 OUTPUT-EBCDIC               PIC X(120).
       01 INPUT-ASCII                 PIC X(120).
       01 OUTPUT-ASCII                PIC X(120).
      * Working Variables
4      01 reply-message               PIC X(120).

      * Procedures Section
       PROCEDURE DIVISION.
       MAINLINE SECTION.
           MOVE 'EXER3A'                     TO register-name.
           MOVE 'S1CELL'                     TO daemongroup.
           MOVE 'S1NODEC'                    TO node-name.
           MOVE 'S1SR01C'                    TO server-name.
5          MOVE 'ServiceName'                TO SRV-service-name.
           MOVE 'This is my reply back!'     TO reply-message.

           INSPECT daemongroup CONVERTING ' ' to LOW-VALUES.

           CALL 'BBOA1REG' USING
                 daemongroup,
                 node-name,
                 server-name,
                 register-name,
                 minconn,
                 maxconn,
                 regopts,
                 rc,
                 rsn.
```

```
           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1REG problem -- rc/rsn : " rc "/" rsn
             GO TO Bad-RC
           ELSE
             DISPLAY "Successfully registered into " daemongroup
           END-IF.

           MOVE LENGTH OF SRV-rqst-area TO SRV-rqst-area-length.
           SET SRV-rqst-area-addr TO ADDRESS OF SRV-rqst-area.
           INSPECT SRV-service-name CONVERTING ' ' to LOW-VALUES.

           CALL 'BBOA1SRV' USING
               register-name,
6          SRV-service-name,
               SRV-service-name-length,
               SRV-rqst-area-addr,
               SRV-rqst-area-length,
               connect-handle,
               wait-time,
               rc,
               rsn,
               rv.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1SRV problem, rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           END-IF.

       * Data conversion and setup for SRP

           DISPLAY "Your message in original ASCII: " SRV-rqst-area.
           MOVE SRV-rqst-area TO INPUT-ASCII.
7          PERFORM ASCII-TO-EBCDIC.
           MOVE OUTPUT-EBCDIC TO SRV-rqst-area.
           DISPLAY "Your message converted to EBCDIC: " SRV-rqst-area.

           DISPLAY "My reply in original EBCDIC: " reply-message.
           MOVE reply-message TO INPUT-EBCDIC.
           PERFORM EBCDIC-TO-ASCII.
           MOVE OUTPUT-ASCII TO SRP-resp-area.
           DISPLAY "My reply converted to ASCII: " SRP-resp-area.

           MOVE LENGTH OF SRP-resp-area TO SRP-resp-area-length.
           SET SRP-resp-area-addr TO ADDRESS OF SRP-resp-area.

           CALL 'BBOA1SRP' USING
               connect-handle,
8          SRP-resp-area-addr,
               SRP-resp-area-length,
               rc,
               rsn.

           IF rc > 0 THEN
             DISPLAY "OLA - BBOA1SRP problem, rc/rsn: " rc "/" rsn
             GO TO Bad-RC
           END-IF.

           CALL 'BBOA1CNR' USING
                 connect-handle,
9                rc,
                 rsn.
```

```
            IF rc > 0 THEN
              DISPLAY "OLA - BBOA1CNR problem, rc/rsn: " rc "/" rsn
              GO TO Bad-RC
            END-IF.

            CALL 'BBOA1URG' USING
                register-name,
                urgopts,
                rc,
                rsn.

            IF rc > 0 THEN
              DISPLAY "OLA - BBOA1URG problem -- rc/rsn: " rc "/" rsn
              GO TO Bad-RC
            ELSE
              DISPLAY "Successfully unregistered from " daemongroup
            END-IF.

            GOBACK.

          *  Code Page Conversions

           EBCDIC-to-ASCII.
               MOVE FUNCTION DISPLAY-OF
10             (FUNCTION NATIONAL-OF (INPUT-EBCDIC EBCDIC-CCSID),
                   ASCII-CCSID)
               TO OUTPUT-ASCII.

           ASCII-to-EBCDIC.
               MOVE FUNCTION DISPLAY-OF
               (FUNCTION NATIONAL-OF (INPUT-ASCII ASCII-CCSID),
                   EBCDIC-CCSID)
               TO OUTPUT-EBCDIC.

          *  Section used to exit batch if any API returned RC>0

           Bad-RC.
               DISPLAY "OLA - EXITING program due to non-RC=0."
               GOBACK.

/*
```

Notes:

1. Registration data values same as before.
2. The `BBOA1SRV` and `BBOA1SRP` APIs have very similar names and purposes. We prefix them with SRV and SRP to making the illustration a bit easier to follow.
3. Data values used to translate ASCII and EBCDIC. The routine to do that is block **10**.
4. Setting up a fixed reply message rather than simply echoing back what came from WAS.
5. The service name that will be used by `BBOA1SRV`. A specific string means the Java code has to name that specific string. An asterisk would mean `BBOA1SRV` listens for *any* service name request coming in on the named registration.
6. The `BBOA1SRV` API is called. Note `connect-handle` ... that is an *output* value you'll use on the `BBOA1SRP` to send a response back.
7. Some housekeeping to convert ASCII / EBCDIC and set up the `BBOA1SRP` call back.
8. The `BBOA1SRP` API is called ... this sends a response back to the Java caller.
9. We release the connection using `BBOA1CNR` and the connect-handle returned on `BBOA1SRV`.
10. The EBCDIC / ASCII conversion routines.

Perform Exercise 3a

In prep for the upcoming exercises, make certain you have:

- ☐ Access to a COBOL compiler and system link editor
- ☐ The WOLA modules copied out[37] to a load library you can reference from LKED SYSLIB
- ☐ A WAS z/OS server environment created and operational and enabled for WOLA[38].

Then:

- ☐ Copy the sample exercise to your system, modify the contents to match your compiler settings, change the cell, node and server short names and compile..

Then:

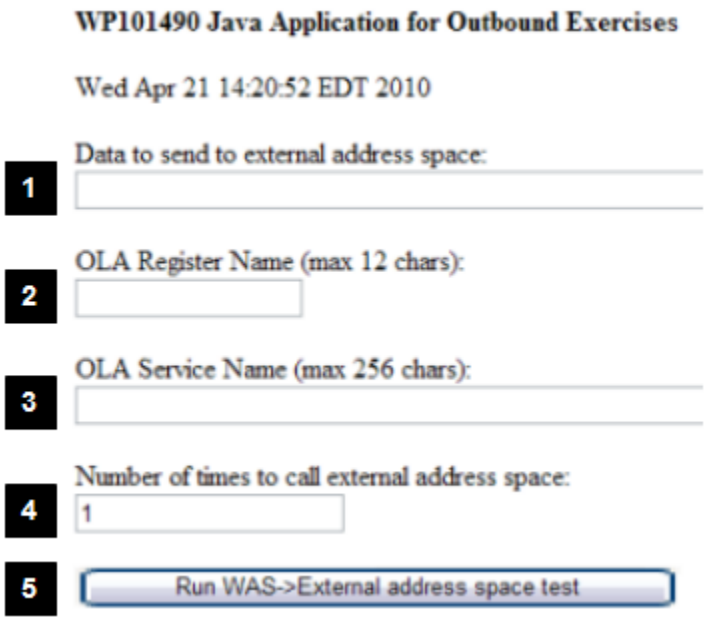
- ☐ Install the **`ATSsample1.ear`** app[39] supplied in the ZIP that accompanies this Techdoc.

  It is a simple application and should be simple to install. There is one resource reference and that should be mapped to your WOLA resource adapter. The JNDI name for that would be eis/ola if you followed the examples for installation. Be sure to save and synchronize.

- ☐ Once installed, start the sample `ATSSample` application.
- ☐ Access the application with URL:

  `http://<host>:<port>/ATSSample1Web/`

- ☐ You'll see the following screen:



Notes:

1. Data to send to the batch COBOL that's hosting the service
2. The register name used on the `BBOA1REG` API of the exercise[40]

37 The `olaInstall.sh` does that for you. InfoCenter search: `tdat_enableconnector` for the syntax.
38 InfoCenter: `tdat_enableconnector`. We covered this in summary on page 18.
39 See page 45 for an important note about which sample app to use, ATSsample1 or ATSSample1-new.
40 For Exercise 3a the value supplied with the sample is `EXER3A`

3. The service name used on the BBOA1SRV API of the exercise[41]
4. The number of times you want the Java program to loop and invoke the COBOL program[42]
5. The button used to send the request over WOLA to the COBOL program.

☐ Fill in the values and click the button at the bottom of the page.

☐ On the browser screen you should see the following:

Output: This is my reply back!
Test Executed

☐ You'll find that the batch program will have ended (no loop).

☐ In the batch program SYSOUT you should see:

```
Successfully registered into S1CELL
Your message in original ASCII:       / /  ?      >/% /

Your message converted to EBCDIC: Test data to external a/s

My reply in original EBCDIC: This is my reply back!

My reply converted to ASCII:          _`    %`  / ,

Successfully unregistered from S1CELL
```

Run the scenario again, but do the following:

☐ Update the source COBOL and change the value of SRV-service-name to an asterisk (*). Keep it in the single quotes to maintain it as a string.

☐ Recompile and re-run the program

☐ From the browser, use the proper register name but use any string (or no string at all) on the service name field. You should see the same results. Note how the service name as an asterisk means the service name used from the Java side is no longer so critical.

Wrap-up of Exercise 3a

BBOA1SRV is a way to "host a service" so that Java programs in WAS z/OS may call out to the service. It provides the "something" for the Java program to talk to[43].

BBOA1SRV is another one of those APIs that packages "primitive" APIs under the covers. SRV is simple to use, but it lacks some of the granular flexibility of the primitives.

We'll begin to look at the primitives beginning with Exercise 3c. First we'll wrapper a loop inside Exercise 3a to see how this function could be used multiple times.

Overview of Exercise 3b - BBOA1SRV, BBOA1SRP with loop

Putting a loop around this sequence of APIs doesn't really change the API usage at all. For that reason we won't show the whole source in this document. But we will note a few things:

- The key thing is the placement of the loop. It starts just before BBOA1SRV and ends just after BBOA1CNR.
- We added a loop counter string to the front-end of the response so proof of it actually looping is provided in the feedback on the browser.

41 For Exercise 3a the value supplied with the sample is ServiceName

42 Use 1 for exercises with no loop (3a); or some reasonable number for programs that have a loop (3b)

43 Reminder: If CICS is the external address space you have the option of avoiding coding any of this. The supplied BBO$ link server task implements these APIs in a way that "hides" it all from you and your programs.

- We added an EVALUATE function to check if the following strings sent:
  - If the string **`STOP`** is sent, then a flag is set and the COBOL `PERFORM` loop is exited.
  - If the string **`RESET`** is sent, the loop counter is set back to `0`.

Perform Exercise 3b

Similar to Exercise 3a:

- ☐ Copy the sample to your system; update and compile
- ☐ From the browser, leave the "Number of times to call external address space" set at 1.
- ☐ Type in a string to send over and then click the button.
- ☐ Notice the response back has a loop counter pre-pended.
- ☐ Use the browser "back" button to return to the input screen.
- ☐ Do that as many times as you'd like. Issue `RESET` to reset the loop counter back to `0`; issue `STOP` to exit the loop and stop the batch program.

Perform a loop using the programmatic option on the Java side:

- ☐ Restart the Exercise 3b batch program.
- ☐ From the browser screen set the number of interations the Java program perform[44].

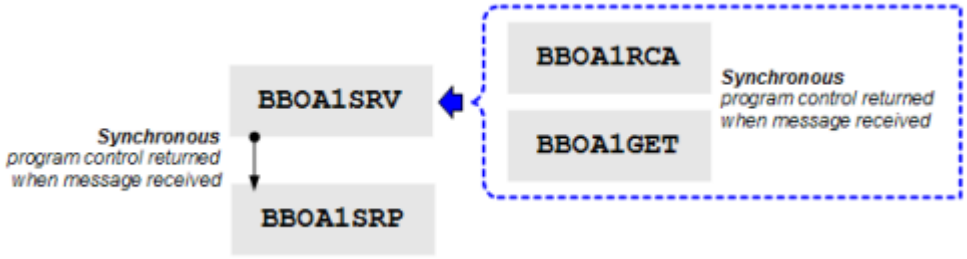
Wrap-up of Exercise 3b

Essentially the same wrap-up as 3a except we put a loop into the mix.

It's now time to leave `BBOA1SRV` and move to the primitive APIs that "host a service"

Beyond `BBOA1SRV` -- the primitive `BBOA1RCA`

As we mentioned earlier, the `BBOA1SRV` API is one of those APIs that packages some "primitives" under the covers. Here's a picture illustrating that:



There's a subtle distinction going on between the `BBOA1SRV` and the more primitive `BBOA1RCA`. In summary:

- One of `BBOA1SRV`'s input parameters is: "`requestdatalen` ***(input)*** *-- A 32-bit unsigned value containing the length of the data area to receive the message into.*" That means you must know at least the *maximum* message length coming.
- `BBOA1RCA`, on the other hand, has "`requestdatalen` ***(output)*** *-- A 32-bit unsigned value is returned containing the length of the data to receive.*"

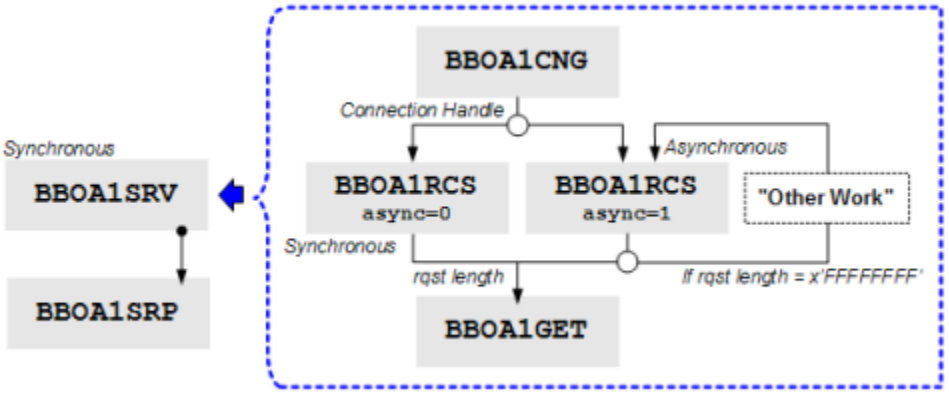
`BBOA1RCA` provides the *actual length* of the message received that you may use on a subsequent `BBOA1GET` API. It also returns the connection-handle on which the message arrived from WAS. That too is used on `BBOA1GET`.

44 Temper your initial enthusiasm to slam in a *really* large number. Start with something reasonable, then work up. ☺

The "A" at the end of RCA means "any" -- BBOA1RCA will accept an outbound message on whatever connection gets used by WOLA[45].

Beyond BBOA1RCA -- an even more "primitive" primative: BBOA1RCS

BBOA1RCS takes this concept one step further:



The synchronous route we'll illustrate with Exercises 3e and 3f.

The asynchronous option we will not illustrate. We discuss this under "What about asynchronous BBOA1RCS?" on page 53.

The point here is really that BBOA1RCS is even more fine-grained than BBOA1RCA. It requires you to issue a "get connection" (BBOA1CNG), which then allows you to call BBOA1RCS to listen on the specific connection received.

The practical distinctions between BBOA1RCA and BBOA1RCS are very subtle. Generally speaking[46] the BBOA1RCS API should be reserved for very specific cases. And then only if you've mastered the other APIs.

Overview of Exercise 3c - BBOA1RCA

This turns out to be very similar to Exercise 3a, which used BBOA1SRV. BBOA1RCA took the place of BBOA1SRV, a BBOA1GET was inserted just before the BBOA1RCA, and the SRV* variables were renamed to RCA*[47]. The program was compiled and run.

Perform Exercise 3c

At this point you likely have the drill down well:

- ☐ Copy in the supplied exercise sample
- ☐ Update and compile
- ☐ Submit
- ☐ From the http://<host>:<port>/ATSSample1Web/ URL update the register and service name fields and invoke with a loop value of 1. This exercise does not loop on the COBOL side, so don't have it loop on the Java side.
- ☐ With one invocation the COBOL program will return a message and then exit.

45 With BBOA1RCS the "S" means "specific" -- you must get a connection with BBOA1CNG first. That's Exercise 3e.
46 This author's opinion.
47 Not strictly required, but it helped cut down on the confusion factor.

Overview of Exercise 3d - BBOA1RCA with a loop

Exercise 3d is simply 3c with the same loop architecture that Exercise 3b had.

Perform Exercise 3d

☐ Copy, update, compile and run.

Overview of Exercise 3e and 3f - synchronous BBOA1RCS and with a loop

Exercise 3e looks very similar to 3c with the following exceptions:

- BBOA1RCA is swapped out in favor of BBOA1RCS.
- A new variable RCSasync was created and set to 0.
- Variables named RCA* were simply renamed to RCS*[48]
- We inserted a BBOA1CNG prior to the issuance of BBOA1RCS.

For Exercise 3f, which added the same loop function seen in 3b and 3d, except we put it *inside* the BBOA1CNG / BBOACNR structure[49].

Perform Exercise 3e and 3f

☐ Copy, update, compile and run.

What about asynchronous BBOA1RCS?

To be quite honest, the author ran to the limit of his COBOL programming skills on this.

The problem was this -- calling BBOA1RCS with async=1 was easy enough. The program then had to do "other work" while the author went over to the browser to set up the outbound call to the BBOA1RCS hosted service. If COBOL had a built-in "delay" or "sleep" function this would have been fairly easy. But it doesn't[50]. So the author -- flailing desperately ☺ -- created several different versions of loops that tried to delay in some fashion. But they all turned out to be rather tight loops. By the time he was able to get back to the browser to set up the call, the COBOL program had chewed up a bunch of CPU and spit out a lot of messages.

At the end of the day the author felt that the objectives of the Techdoc had been met. So rather than pursue it further he wrapped up the document.

Wrap-up of the outbound exercises

We explored three more APIs -- BBOA1SRV, BBOA1RCA and BBOA1RCS. All three "host a service" but do so with increasing degrees of granular control.

When the external address space "hosts a service" it puts the program in a state ready to accept a connection and service invocation from Java in WAS z/OS.

Note: If going outbound to CICS, then you may not need to code to these APIs. The supplied BBO$ link server task does that. But ... if you really want to squeeze out the maximum performance then you may want to consider coding to these APIs even if in CICS. The BBO$ link server task is wonderfully easy, but it represents a degree of overhead your requirements may not permit.

---

48 Again, not strictly required, but it helped cut down on the confusion factor.

49 Since BBOA1RCS does not itself do a BBOA1CNG under the covers, we had the opportunity to secure a connection and re-use it over and over again. BBOA1SRV and BBOA1RCA perform a CNG under the covers, and if you don't release it with each loop you'll quickly exhaust the maximum connections, which is set on BBOA1REG.

50 CICS does, so BBOA1RCS async=1 in CICS might be easier to code to. And it's possible to code a C stub with a sleep-type function and call that from COBOL. But at this point the author decided the merits of showing BBOA1RCS async=1 didn't justify the effort. The *concept* of asynchronous WOLA was illustrated earlier in the inbound call exercises. We'll leave it at that.

Appendix - Miscellaneous Information

In this section we'll provide some supporting information with pointers to InfoCenter articles where the full detail is provided.

Quick checklist for enabling the WAS environment for these exercises

Note: This is just for enabling the WAS server environment for these exercises. If you're using CICS then there is additional setup within CICS. See the InfoCenter for more (search on `tdat_enableconnectorcics`).

- ☐ Have a server environment ready ... either ND or Standalone

  Either will work for the purposes of these exercises.

- ☐ Make sure *at least* V7.0.0.4 is applied

  Version 7.0.0.4 is the maintenance level that first introduced WOLA. You must have at least that level. Make sure that `applyPTF.sh` has run against the node so that 7.0.0.4 (or higher) has been applied to the node.

- ☐ Use `olaInstall.sh` to enable the node for WOLA

  In the InfoCenter, search on `tdat_enableconnector`. That page spells out how to use the `olaInstall.sh` shell script to enable a node to use WOLA.

- ☐ Have at least load module library created with the WOLA modules copied from the HFS

  One of the options of `olaInstall.sh` is to copy the WOLA modules from the HFS to a pre-allocated load module library. The InfoCenter page `tdat_enableconnector` spells out the details.

- ☐ Install the `ola.rar` resource adapter in the node

  This is located in the /<smpe_root>/mso/OLA/installableApps/ directory for your WAS V7 product installation. It installs like any other JCA adapter. Create a connection factory with a JNDI name of `eis/ola`

- ☐ Set the `WAS_DAEMON_ONLY_enable_adapter` cell-level environment variable

  That should be scoped to the "cell" level and the value you provide it should be `true`.

- ☐ Install the `OLASample1.ear` file

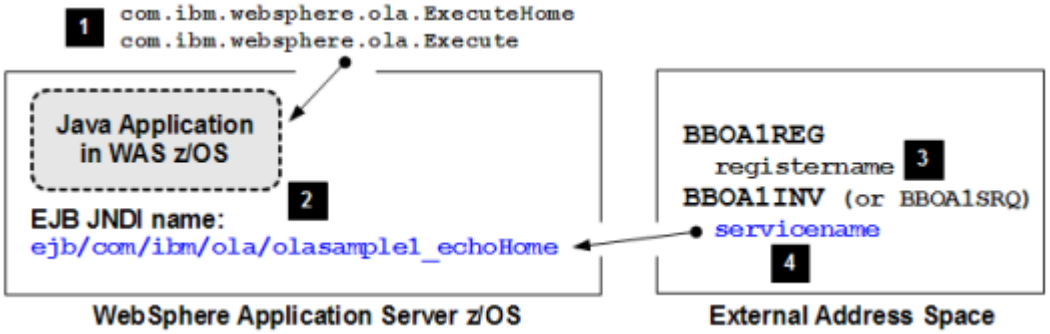
  That should be scoped to the "cell" level and the

- ☐ Provide access to `CB.BIND.<prefix>.**` profile for IDs wishing to access WAS server

- ☐ Restart the entire environment

A picture representation of the relationships when using WOLA

Inbound to WAS from external address space (Batch or CICS)

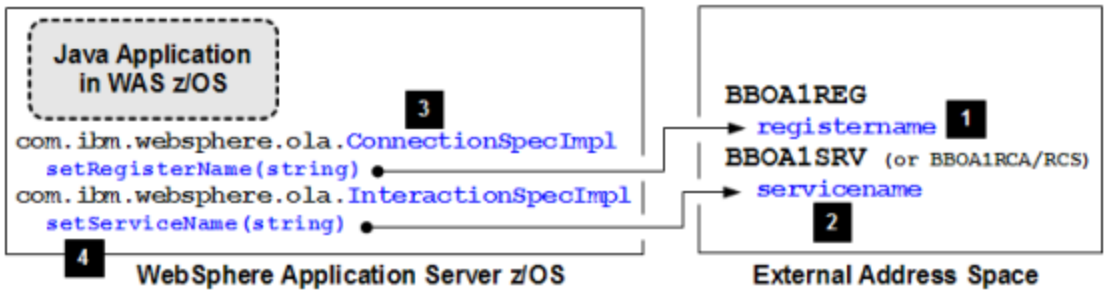


Notes:

1. The target for the inbound call *must* be implemented as a stateless session bean, and it *must* implement its `ExecuteHome` and `Execute` interfaces using the `com.ibm.websphere.ola` package.
2. When that EJB is deployed into the server, it is given a JNDI name for the home interface of the bean.
3. The program in the external address spaces uses the `BBOA1REG` API to register and it provides a register name. That may be whatever you wish provided it is exactly 12 characters long and is blank padded.
4. The program wishing to call the EJB uses the `BBOA1INV` API (or `BBOA1SRQ` as we illustrated earlier) and provides the EJB's JNDI name as the `servicename` on the API.

> **Key:** The `BBOA1REG` API establishes the linkage to the server[51]. The `BBOA1INV` names the particular EJB to invoke by providing the JNDI name as the "service name."

Outbound from WAS to external Batch (or USS)



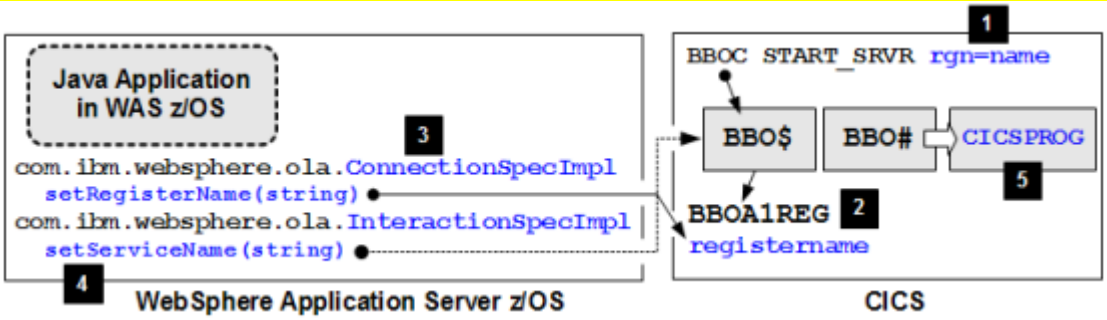
Notes:

1. The external program uses `BBOA1REG` and provides a register name.
2. The external program "hosts a service" and provides a service name.
3. The Java program uses `ConnectionSpecImpl` and `setRegisterName()` to specify the registration pool to connect to.
4. The Java program uses `InteractionSpecImpl` and `setServiceName()` to specify the service within the registration pool it has connected to.

> **Key:** If the `BBOA1SRV` API specifies an explicit servicename, then the Java program must know and use the service name specified[52]. However, if `BBOA1SRV` uses an asterisk for the service name, then it will accept requests on *any service on that registration*.

51 Recall that the `BBOA1REG` API requires the cell, node *and server* short names. The registration is specific to the server.
52 When outbound to CICS with the `BBO$` link server task in use, the service name is the CICS program name to invoke.

Outbound from WAS to CICS



Notes:

1. The `BBOC`[53] control transaction is used to start the `BBO$` link server task with rgn= being the registration name that will be used.
2. That issues the `BBOA1REG` API "under the covers".
3. Java program uses `ConnectionSpecImpl` and specifies the registration name used.
4. Java program uses `InteractionSpecImpl` and specifies the CICS program name to invoke.
5. The request flows through `BBO$`/`BBO#` and an EXEC CICS LINK is performed against the program.

> **Key:** Even though the `BBO$` link server task is used, there's still a registration that takes place, and the Java program still needs to know what that value is. The service name is simply the CICS program name that will be issued on the EXEC CICS LINK.

53 InfoCenter, search on `rdat_cics`

Document Change History

Check the date in the footer of the document for the version of the document.

Apr 27, 2010	Original document.
Sept 12, 2010	Updated to reference the IMS support made available in 7.0.0.12.
Dec 14, 2010	Fixed a serious problem with the `WORKING STORAGE` sections in the samples for the asynchronous flags. `PIC 9(1) COMP 0.` ⇦ Incorrect, and caused asynch behavior problems `PIC 9(8) COMP 0.` ⇦ Corrected The accompanying samples in the ZIP file have been updated as well.
Aug 13, 2013	Added a note about the use of the ATSsample1.ear application compared to ATSSample1-new.ear. The difference is a transaction attribute of `NOT_SUPPORTED` in the "new" version of that app. The README in the ZIP file suggests the "new" is for V8 but we have found it may be required for V7 as well. The symptom of the problem is an error with a minor code of C9C2C3B.

End of WP101490