



isCOBOL™ EIS
isCOBOL Enterprise Information System

Copyright © 2014 Veryant LLC.

All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution and recompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Veryant and its licensors, if any.

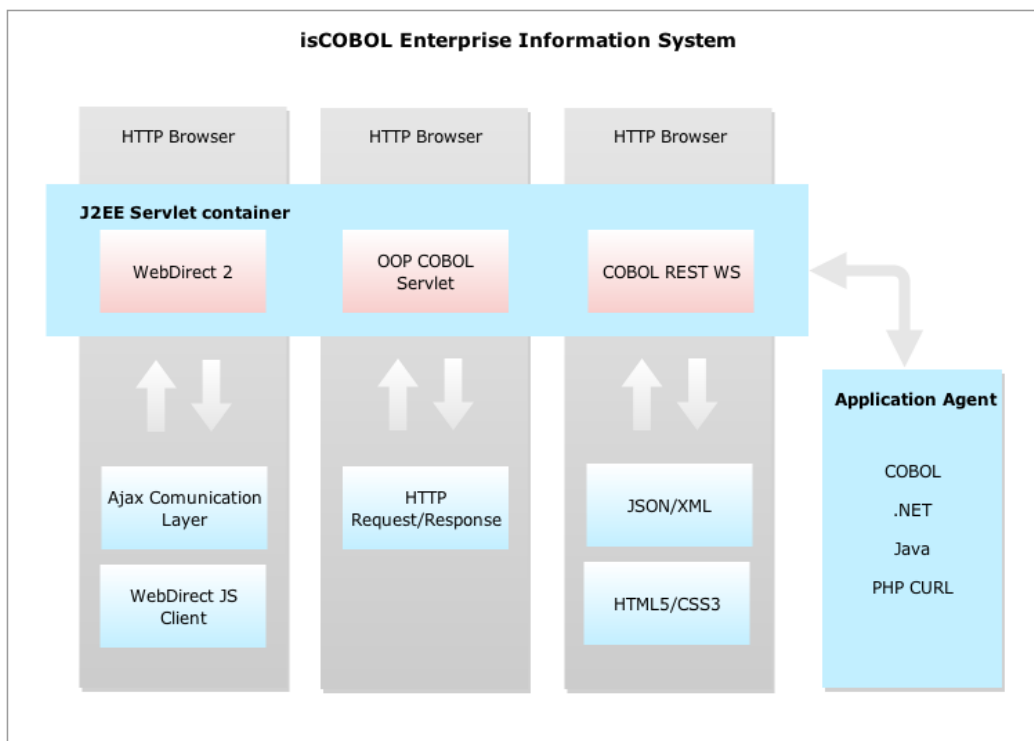
Veryant and isCOBOL are trademarks or registered trademarks of Veryant LLC in the U.S. and other countries. All other marks are property of their respective owners.

isCOBOL Enterprise Information System 2014 Release 1

Introduction

isCOBOL Enterprise Information System (EIS) is an umbrella of tools and features available in the isCOBOL Evolve Suite that allows development and execution of a web based application in a J2EE container. There are several options to deploy a web application based on EIS as shown in Figure 1, *isCOBOL Enterprise Information System Architecture*, in order to provide the right option for every scenario.

Figure 1. isCOBOL Enterprise Information System Architecture



isCOBOL EIS (Enterprise Information System), Web Service option

A Web Service is a software system designed to support interoperable machine-to-machine interaction over a network.

Many organizations use multiple software systems for management. Different software systems often need to exchange data with each other, and a web service is a method of communication that allows two software systems to exchange this data over the internet. The software system that requests data is called a *service requester* or *consumer*, whereas the software system that would process the request and provide the data is called a *service provider* or *producer*.

Different software might be built using different programming languages, and hence there is a need for a method of data exchange that doesn't depend upon a particular programming language. Most types of software can, however, interpret **XML** or **JSON** tags. Thus, web services can use **XML** or **JSON** files for data exchange.

Two predominant web services frameworks, **REST** and **SOAP**, are used in web site development.

REST, *Representational State Transfer* and **SOAP**, *Simple Object Access Protocol*, provide mechanisms for requesting information from resources, **REST**, or from endpoints, **SOAP**. Perhaps the best way to think of these technologies is as a method of making a remote procedure calls against a well-defined API. **SOAP** has a more formal definition mechanism called **WSDL**, *Web Services Definition Language*, and is more complex to implement. **REST** uses the standard **HTTP** request and response mechanism, simplifying implementation and providing for a more flexible, loose coupling of the client and server. Note that **REST** also supports the transfer of non-XML messages such as **JSON**, *JavaScript Object Notation*.

COBOL approach using REST

As explained above, there is software that is called a *service requester* or *consumer*, and there is software that is called a *service provider* or *producer*.

COBOL REST producer

In order to develop a **COBOL REST producer** (server- side), to process requests and provide data, the COBOL program has to be transformed to be executed like a Web Service *REST*. This objective is achieved through the **HTTPHandler** class that allows communication with HTML pages through AJAX retrieving data and printing results.

In the **isCOBOL** sample folder you will find the folder `eis/webservices/rest` that contains an example of a COBOL REST producer (REST Web Service) and an example of a COBOL REST consumer to be used to test the service.

In the server folder there is a COBOL source program called **ISFUNCTIONS.cbl** that exposes two services: `ISFUNCTION_GETZIP` and `ISFUNCTION_GETCITY` that allow searching a US city name by zip code or by name.

This program has three entries:

- **MAIN**, the default entry where the values to be used are loaded from the JSON stream:

```
move "94101"           to a-zipcode(1) .
move "San Francisco"  to a-city(1) .
move "San Francisco"  to a-county(1) .
move "California"     to a-state(1) .

move "92123"           to a-zipcode(2) .
move "San Diego"      to a-city(2) .
move "San Diego"      to a-county(2) .
move "California"     to a-state(2) .

move "10001"           to a-zipcode(3) .
move "New York"       to a-city(3) .
move "New York"       to a-county(3) .
move "New York"       to a-state(3) .

move "89044"           to a-zipcode(4) .
move "Las Vegas"      to a-city(4) .
move "Clark"          to a-county(4) .
move "Nevada York"    to a-state(4) .

move "Program Loaded" to ok-message;;
comm-area:>displayJSON (ok-page) .
goback.
```

- ISFUNCTION_GETZIP, a COBOL entry point that receives into *isfunction-getZipCode* working storage structure, a name of a US city and returns the zip code into *isfunction-returnZipCode* as JSON stream using displayJSON() method:

```
entry "ISFUNCTION_GETZIP" using comm-area.

comm-area:>accept (isfunction-getZipCode).

move 1 to idx.
search array-data varying idx
  at end
    move "Zip code not Found" to returnZipCode
    when city-zipCode = a-city(idx)
      move a-zipcode(idx) to returnZipCode
  end-search.

comm-area:>displayJSON (isfunction-returnZipCode).

goback.
```

where *isfunction-getZipCode* working storage structure is defined like:

```
01 isfunction-getZipCode identified by "".
03 identified by "get_Zip_Code".
05 city-zipCode pic x any length.
```

and *isfunction-returnZipCode* working storage structure is defined like:

```
01 isfunction-returnZipCode identified by "".
03 identified by "Zip_Code".
05 returnZipCode pic x any length.
```

- ISFUNCTION_GETCITY, a COBOL entry point that receives into *isfunction-getCity* working storage structure, a zip code of a US city and return the city name into *isfunction-receivedCity* variable as JSON stream using displayJSON() method:

```
entry "ISFUNCTION_GETCITY" using comm-area.

comm-area:>accept (isfunction-getCity).

move 1 to idx.
search array-data varying idx
  at end
    move "City not Found" to returnCity
    when zipCode-city(1:5) = a-zipcode(idx)
      move a-city(idx) to returnCity
  end-search.

comm-area:>displayJSON (isfunction-recivedCity).

goback.
```

where *isfunction-getCity* working storage structure is defined like:

```
01 isfunction-getcity identified by "".  
03 identified by "get_City".  
05 zipCode-city pic x any length.
```

and *isfunction-receivedCity* working storage structure is defined like:

```
01 isfunction-recivedCity identified by "".  
03 identified by "City".  
05 returnCity pic x any length.
```

In order to have this ISFUNCTIONS.cbl working correctly, it should be compiled using isCOBOL 2014R1 compiler and deployed inside a Java Servlet container like Tomcat, JBOSS IBM WebSphere or BEA WebLogic.

COBOL REST consumer

In order to develop a **COBOL REST consumer** (client-side), to invoke REST Web Service, the COBOL program should take advantage of **HTTPClient** class that allows to communicate with **COBOL REST producer** entry points through **HTTP** protocol. Also to allow definition of HTTP parameters, an **HTTPData.Params** class is provided.

In the **isCOBOL** sample folder you find the folder `eis/webservices/rest/client` that contains an example of COBOL client program called **CLIENTH.cbl** of previous ISFUNCTIONS.cbl server service.

This program has the objective to invoke ISFUNCTION_GETZIP service and ISFUNCTION_GETCITY to have the zip code of San Diego and to have the name of the city whose zip code is 89044.

This program must take the following steps:

- Include HTTPClient and HTTPData.Params classes in COBOL repository:

```
configuration section.  
repository.  
  class http-client as "com.iscobol.rts.HTTPClient"  
  class http-param as "com.iscobol.rts.HTTPData.Params".
```

- Establish the connection with REST Web Service using `doGet()` method and checking the success of the operation using `getResponseCode()` method:

```
http:>doGet ("http://127.0.0.1:8080/isfunctions/servlet/isCobol (ISFUNCTIONS)")  
http:>getResponseCode (response-code)
```

- Prepare the city name as parameter to be pass to the service

```
move "San Diego" to city-zipCode.  
set params = http-param:>new()>add("get_Zip_Code", city-zipCode).
```

- Invoke the ISFUNCTION_GETZIP with prepared parameter and getting back the zip code:

```
http:>doGet ("http://127.0.0.1:8080/isfunctions/"  
           "servlet/isCobol (ISFUNCTION_GETZIP)", params)  
  
http:>getResponseCode (response-code)  
if response-code = 200  
  http:>getResponseJSON (isfunction-recivedZipCode)
```

where `isfunction-recivedZipCode` working storage structure is defined like:

```
01 isfunction-recivedZipCode identified by "".  
03 identified by "Zip_Code".  
05 zipCode pic x any length.
```

and 92123 is the zip code of San Diego saved into `zipCode` COBOL variable.

A similar approach should be the following, have the city name provide the zip code.

In order to have this CLIENTH.cbl working correctly, it should be compiled using isCOBOL 2014R1 compiler and run using isCOBOL EIS 2014R1:

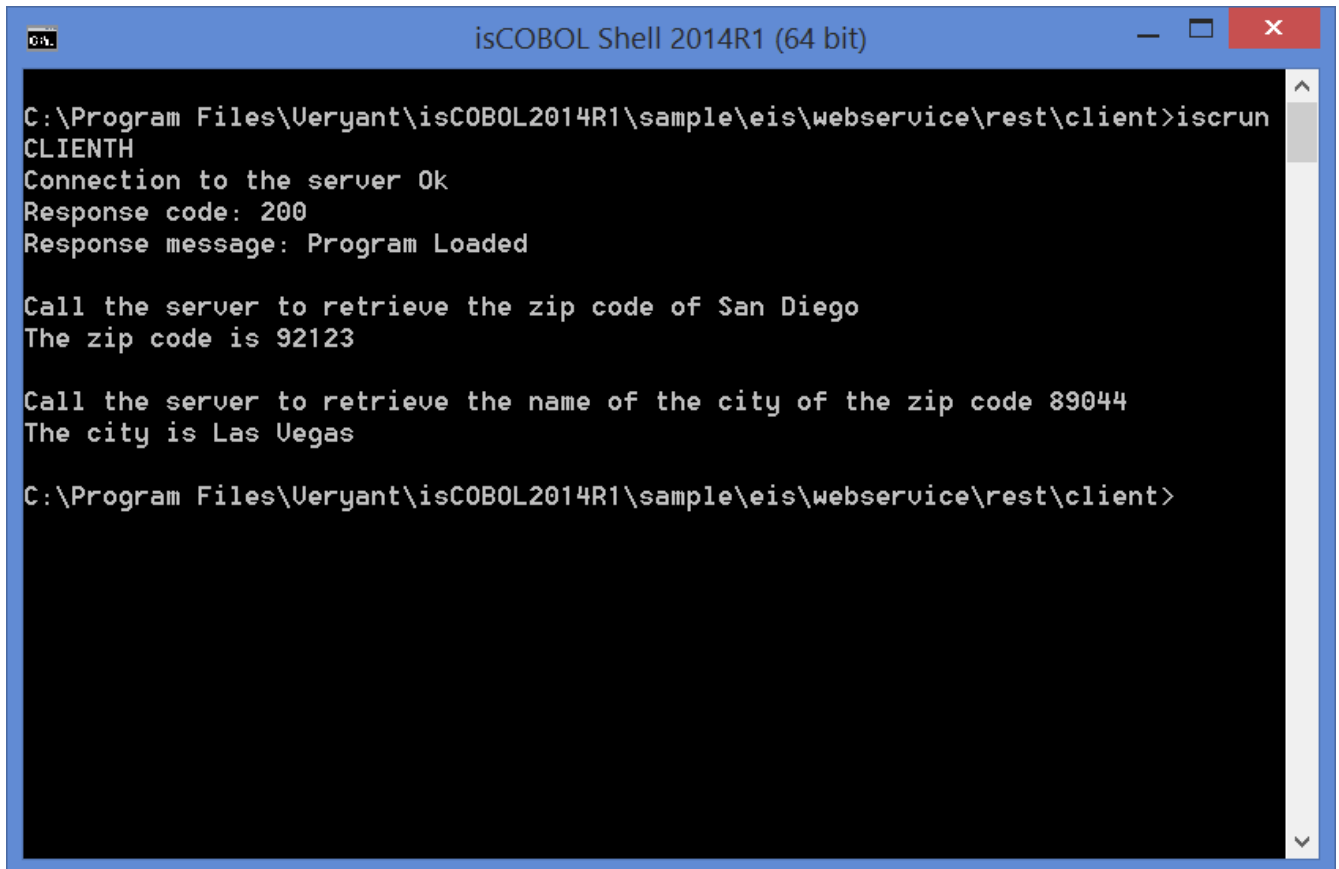
Compile the program with the command:

```
iscc clienth.cbl
```

and run it with the command:

```
isrun CLIENTH
```

this is the result:



```
isCOBOL Shell 2014R1 (64 bit)
C:\Program Files\Veryant\isCOBOL2014R1\sample\eis\webservice\rest\client>isrun
CLIENTH
Connection to the server Ok
Response code: 200
Response message: Program Loaded

Call the server to retrieve the zip code of San Diego
The zip code is 92123

Call the server to retrieve the name of the city of the zip code 89044
The city is Las Vegas

C:\Program Files\Veryant\isCOBOL2014R1\sample\eis\webservice\rest\client>
```

COBOL approach using SOAP

Starting with isCOBOL version 2014R1, EIS is provided as preliminary support to develop COBOL programs capable to consume SOAP Web Services based on XML.

COBOL SOAP consumer

In order to develop a **COBOL SOAP consumer** (client-side), to invoke SOAP Web Service, the COBOL program should take advantage of **HTTPClient** class. That class contains several useful methods to work with SOAP Web Service.

In the **isCOBOL** sample folder you find the folder `eis/webservices/soap/client` that contains an example of COBOL client program called **IP2GEO.cbl** that shows how to use a SOAP Web Service available over internet at <http://ws.cdyne.com/ip2geo/ip2geo.asmx>.

This service "Resolves IP addresses to Organization, Country, City, and State/Province, Latitude, Longitude. In most US cities it will also provide some extra information such as Area Code, and more."

A SOAP Web Service usually provides a way to inquire the functionality available and the parameters that should be used. To simplify the working storage definition able to manage the XML envelope, a new utility called **WSDL2Wrk** is provided.

That utility is able to read WSDL definition obtained adding ?WSDL to the Web Service URL definition, something like: <http://ws.cdyne.com/ip2geo/ip2geo.asmx?WSDL> generates the following working storage :

```
*> binding name=IP2GeoSoap, style=  
01 soap-ResolveIP-input identified by 'Envelope'  
   namespace 'http://www.w3.org/2003/05/soap-envelope'.  
03 identified by 'Body'.  
   06 identified by 'ResolveIP'  
     namespace 'http://ws.cdyne.com/'.  
   07 identified by 'ipAddress'.  
     08 a-ipAddress pic x any length.  
   07 identified by 'licenseKey'.  
     08 a-licenseKey pic x any length.
```

```
01 soap-ResolveIP-output identified by 'Envelope'  
   namespace 'http://www.w3.org/2003/05/soap-envelope'.  
03 identified by 'Body'.  
   06 identified by 'ResolveIPResponse'  
      namespace 'http://ws.cdyne.com/'.  
   07 identified by 'ResolveIPResult'.  
   08 identified by 'City'.  
   09 a-City pic x any length.  
   08 identified by 'StateProvince'.  
   09 a-StateProvince pic x any length.  
   08 identified by 'Country'.  
   09 a-Country pic x any length.  
   08 identified by 'Organization'.  
   09 a-Organization pic x any length.  
   08 identified by 'Latitude'.  
   09 a-Latitude pic x any length.  
   08 identified by 'Longitude'.  
   09 a-Longitude pic x any length.  
   08 identified by 'AreaCode'.  
   09 a-AreaCode pic x any length.  
   08 identified by 'TimeZone'.  
   09 a-TimeZone pic x any length.  
   08 identified by 'HasDaylightSavings'.  
   09 a-HasDaylightSavings pic x any length.  
   08 identified by 'Certainty'.  
   09 a-Certainty pic x any length.  
   08 identified by 'RegionName'.  
   09 a-RegionName pic x any length.  
   08 identified by 'CountryCode'.  
   09 a-CountryCode pic x any length.
```

This program has the objective to invoke the **ResolveIP** functionality providing the IP address and receiving some geographic information like City, State, Country, etc.

This program must take the following steps:

- Include HTTPClient class in COBOL repository:

```
configuration section.  
repository.  
  class http-client as "com.iscobol.rts.HTTPClient"
```

- Include the working storage definition to use XML envelope generated from WSDL by WSDL2Wrk utility:

```
WORKING-STORAGE SECTION.  
  copy "ip2geo.cpy".
```

- Provide the IP address to obtain information and call the ResolveIP service using doPostEx() method passing the URL of service, the SOAP media type and the input envelope generated from WSDL2Wrk for ResolveIP service:

```
move "209.235.175.10" to a-ipAddress  
http:>doPostEx (  
  "http://ws.cdyne.com/ip2geo/ip2geo.asmx"  
  "application/soap+xml; charset=utf-8"  
  soap-ResolveIP-input).
```

- check the response if successful and show results:

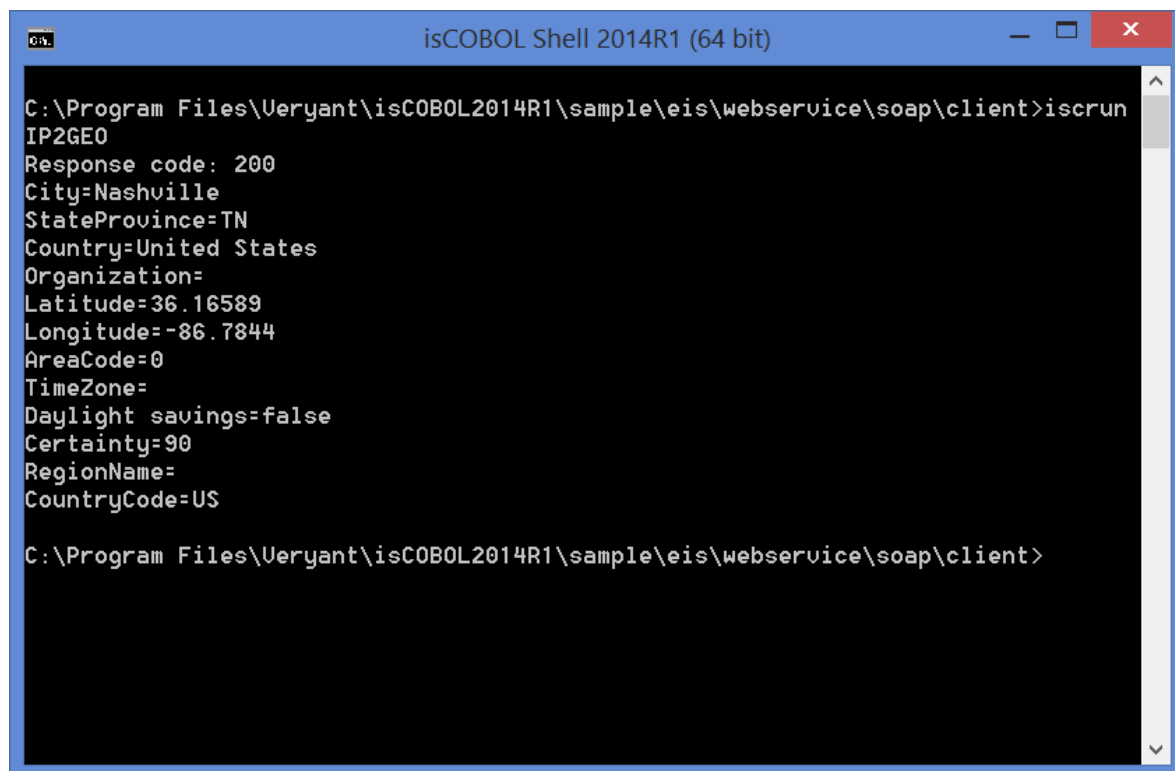
```
http:>getResponseCode (response-code).
display "Response code: " response-code.
if response-code = 200
  http:>getResponseXML (soap-ResolveIP-output)
  display "City=" a-city
  display "StateProvince=" a-stateProvince
  display "Country=" a-country
  display "Organization=" a-Organization
  display "Latitude=" a-latitude
  display "Longitude=" a-longitude
  display "AreaCode=" a-areaCode
  display "TimeZone=" a-timeZone
  display "Daylight savings=" a-HasDaylightSavings
  display "Certainty=" a-certainty
  display "RegionName=" a-regionName
  display "CountryCode=" a-countryCode
```

In order to have this IP2GEO.cbl program working correctly, it should be compiled using isCOBOL 2014R1 compiler and run using isCOBOL EIS 2014R1:

Compile the program with the command:

```
iscc IP2GEO.cbl
```

This is the result of execution of IP2GEO that consumes the **ResolveIP** SOAP Web Service:

A screenshot of a Windows command prompt window titled "isCOBOL Shell 2014R1 (64 bit)". The window shows the execution of a program named IP2GEO. The output is as follows:

```
C:\Program Files\Veryant\isCOBOL2014R1\sample\eis\webservice\soap\client>isccrun
IP2GEO
Response code: 200
City=Nashville
StateProvince=TN
Country=United States
Organization=
Latitude=36.16589
Longitude=-86.7844
AreaCode=0
TimeZone=
Daylight savings=false
Certainty=90
RegionName=
CountryCode=US

C:\Program Files\Veryant\isCOBOL2014R1\sample\eis\webservice\soap\client>
```

Authentication and Authorization method

You can obtain limited access to an HTTP Service taking advantage of existing Authentication and Authorizations providers like Google and Facebook based on OAuth 2.0 standard.

OAuth 2.0 is an open protocol to allow secure authorization in a simple and standard method from web, mobile and desktop applications. The request is to make a COBOL program accessible only by the logged users without checking for each single program.

Servlet Container Configuration

Servlet containers (e.g. Apache Tomcat) have fully configurable authentication systems, however they usually don't fit well with the authentication from another server, thus they are not used in this example.

You need to define a safe area where the isCOBOL application can be invoked only after a successful authentication. Since the isCOBOL applications are executed in the same context as if they were belonging to the same session, you can set an environment property after the authentication process and then check for it each time an application runs. However it is not handy nor safe to put a check in each program, thus you can define a filter that does this job.

The configuration file web.xml will therefore contain the following entries:

```
<filter>
  <filter-name>isCOBOL security</filter-name>
  <filter-class>SecurityFilter</filter-class>
</filter>
<filter-mapping>
  <filter-name>isCOBOL security</filter-name>
  <url-pattern>/servlet/*</url-pattern>
</filter-mapping>
```

In this way you specify a program to run before running any program located under the URL /servlet. This program could be the following isCOBOL class:

```
class-id. SecurityFilter as "SecurityFilter"
                                implements c-filter.

configuration section.
repository.
    class j-ioexception as "java.io.IOException"
    class c-filter as "javax.servlet.Filter"
    class c-filter-chain as "javax.servlet.FilterChain"
    class c-filter-config as "javax.servlet.FilterConfig"
    class c-ServletException as "javax.servlet.ServletException"
    class c-ServletRequest as "javax.servlet.HttpServletRequest"
    class c-ServletResponse as "javax.servlet.HttpServletResponse"
    class c-HttpServletResponse as
        "javax.servlet.http.HttpServletResponse"
    class c-HttpServletRequest as
        "javax.servlet.http.HttpServletRequest"
    .

id division.
object.
data division.
working-storage section.

procedure division.

id division.
method-id. init as "init".
linkage section.
77 cfg object reference c-filter-config.
procedure division using cfg raising c-ServletException.
main.
end method.

id division.
method-id. c-destroy as "destroy".
procedure division.
main.
end method.

id division.
method-id. doFilter as "doFilter".
working-storage section.
77 email pic x any length.
77 uri pic x any length.
77 http-response object reference c-HttpServletResponse.
linkage section.
77 request object reference c-ServletRequest.
77 response object reference c-ServletResponse.
77 f-chain object reference c-filter-chain.
procedure division using request response f-chain
                        raising c-ServletException j-IOException.
main.
    accept email from environment "openid.email".
    if email = space
        set http-response to response as c-HttpServletResponse
        http-response:>sendError
            (c-HttpServletResponse:>SC_FORBIDDEN)
    else
        f-chain:>doFilter (request response)
    end-if.
end method.
end object.
```

This program simply checks if the property "openid.email" has been set to a value different from space and in that case it forwards the execution to the next filter in the chain, otherwise it stops the execution with an error code.

This assures you that any program under the URL /servlet, the safe area, will be executed only if previously in the same session, some program has set the property.

You now need to write that program and define it outside the safe area.

Facebook Authentication

Here we show an example of how to implement a program in order to authenticate the access using the Facebook authentication. You can find Facebook's documentation at the address: <https://developers.facebook.com/docs/facebook-login/manually-build-a-login-flow/v2.0>

This kind of authentication requires your program to redirect the login phase to the Facebook site and then performs some HTTP requests to the Facebook APIs. Your program will use the following classes:

```
configuration section.  
repository.  
    class web-area as "com.iscobol.rts.HTTPHandler"  
    class http-client as "com.iscobol.rts.HTTPClient"  
    class http-params as "com.iscobol.rts.HTTPData.Params"  
    class j-bigint as "java.math.BigInteger"  
    class j-securernd as "java.security.SecureRandom"  
    .  
working-storage section.  
01  params object reference http-params.  
01  http object reference http-client.
```

The classes `j-bigint` and `j-securernd` are used to create a secure random number whose purpose will be explained later.

In order to use the Facebook authentication, you need a Facebook App ID that you can create and retrieve on the App Dashboard (<https://developers.facebook.com/apps/>).

There you get a client ID and a client secret that are necessary in the authentication process.

Let's say that the URL of our program is `http://veryant.com/oauth/FBConnect`, then the WORKING-STORAGE SECTION will contain:

```
78  client-id value "<client-id-by-Facebook>".  
78  clsc value "<client-secret-by-Facebook>".  
78  redir value "http://veryant.com/oauth/FBConnect".  
78  realm value "http://veryant.com/oauth".  
01  state pic x any length.
```


The login process can be divided in three stages:

- Request the authentication from Facebook through a redirection;
- Get the authentication data in order to be able to query Facebook APIs;
- Get the logged user data.

The program is called two times: the first time by the user in order to start the authentication process and the second time by a Facebook redirection.

The first phase is simply a redirection in where you specify what URL must be called back.

You must protect the security of your users by preventing request forgery attacks. In order to be sure that this callback is performed by the URL you actually called, a random id (*state token*) must be supplied. According to Google documentation (<https://developers.google.com/accounts/docs/OAuth2Login>): "One good choice for a state token is a string of 30 or so characters constructed using a high-quality random-number generator". These tokens are often referred to as cross-site request forgery (CSRF) tokens.

You can create this secure random id using the classes *j-securernd* and *j-bigint* as in following code :

```
set state=j-bigint:>new(130 j-securernd:>new) :>toString(32).
```

The code for redirection then will be:

```
phase-1-redirection.  
set state=j-bigint:>new(130 j-securernd:>new) :>toString(32).  
set params = http-params:>new  
    :>add ("client_id" client-id)  
    :>add ("display" "popup")  
    :>add ("response_type" "code")  
    :>add ("scope" "email")  
    :>add ("redirect_uri" redir)  
    :>add ("state" state)  
  
comm-area:>redirect (  
    "https://www.facebook.com/dialog/oauth" params).
```

The second phase begins when the same application is called back by Facebook, as specified by the *redir* variable. The program can easily tell if it is the first run or the second by the setting of the variables *state* and *http-state*: the former is set by phase 1 while the latter will be passed by Facebook in the redirection of the login. So the initial part of the program could be the following one:

```
linkage section.
01 comm-area object reference web-area.
procedure division using comm-area.
main.

    accept client-id from environment "app_id_by_fb"
    accept clsc   from environment "app_secret_by_fb".

    accept redir from environment "realdir_fb".

    if user-email = ""
        perform do-auth
    else
        perform run-first-program
    end-if.
    goback.

do-auth.
    initialize http-response.
    comm-area:>accept(http-response).
    if http-state = space
        perform phase-1-redirection
    else
        if http-state = state
            perform phase-2-get-auth-token
            perform phase-3-get-info
            perform set-first-program
            perform run-first-program
        else
            string "Forged state! (" http-state ") (" state ")"
                into err-msg
            comm-area:>displayError(403 err-msg)
        end-if
    end-if.
```

The parameters received by Facebook are described in the following variable:

```
01 http-response identified by "_".
03 identified by "state".
05 http-state   pic x any length.
03 identified by "code".
05 http-code   pic x any length.
```

The parameter *code* (stored in *http-code*) is the one you need in order to get the authorization to query the Facebook APIs, along with your client ID and client secret. The source code of the second phase could be the following:

```
phase-2-get-auth-token.  
  set http = http-client:>new  
  set params = http-params:>new  
    :>add ("code" http-code)  
    :>add ("client_id" client-id)  
    :>add ("client_secret" clsc)  
    :>add ("redirect_uri" redir)  
    :>add ("grant_type" "authorization_code")  
  try  
    http:>doPost (  
      "https://graph.facebook.com/oauth/access_token" params)  
    http:>getResponseCode (response-code)  
    if response-code = 200  
      http:>getResponsePlain (fb-token)  
    else  
      comm-area:>displayError(response-code "")  
      goback  
    end-if  
  catch exception  
    comm-area:>displayError(500 exception-object:>toString)  
    goback  
  end-try.
```

If the request is successful, the program will receive a character string, called "access token", that allows you to call anything among the Facebook APIs. You still don't have any information about the person who is logged, so you need to get some basic information.

In the third phase you may choose to call the API "me": this API returns a JSON payload whose data is described in the following variable:

```
01 user-info identified by "_".  
  03 identified by "id".  
    05 user-id pic x any length.  
  03 identified by "email".  
    05 user-email pic x any length.  
  03 identified by "verified".  
    05 user-verified-email pic x any length.  
  03 identified by "name".  
    05 user-name pic x any length.  
  03 identified by "first_name".  
    05 user-given-name pic x any length.  
  03 identified by "last_name".  
    05 user-family-name pic x any length.  
  03 identified by "link".  
    05 user-link pic x any length.  
  03 identified by "picture".  
    05 user-picture pic x any length.  
  03 identified by "gender".  
    05 user-gender pic x any length.
```

The source code could be the following:

```
phase-3-get-info.  
  string "https://graph.facebook.com/me?" fb-token  
    into authorization  
  set http = http-client:>new  
  try  
    http:>doGet (authorization)  
    http:>getResponseCode (response-code)  
    if response-code = 200  
      http:>getResponseJSON (user-info)  
    else  
      comm-area:>displayError(response-code "")  
      goback  
    end-if  
  catch exception  
    comm-area:>displayError(500 exception-object:>toString)  
    goback  
  end-try.
```

Note that this time there is a STRING command instead of passing the parameters in the usual way. This is because the access token must be passed as it is.

If the call is successful, then the only thing to do is start the next program, i.e. the first program in the application, for example:

```
set-first-program.  
  set environment "openid.email" to user-email.  
  accept data-dir from environment "file.prefix"  
  string data-dir "/" user-email into data-dir  
  
  call "c$makedir" using data-dir  
  set environment "file.prefix" to data-dir.  
  
run-first-program.  
  comm-area:>redirect ("_index.html").
```

Google Authentication

Here we show an example about how to implement a program in order to authenticate the access using Google authentication. You can find Google's documentation at the address: <https://developers.google.com/accounts/docs/OAuth2Login>.

This kind of authentication requires your program to redirect the login phase on the Google site and then performs some HTTP requests to the Google APIs. Your program will use the following classes:

```
configuration section.  
repository.  
  class web-area      as "com.iscobol.rts.HTTPHandler"  
  class http-client  as "com.iscobol.rts.HTTPClient"  
  class http-params  as "com.iscobol.rts.HTTPData.Params"  
  class j-bigint     as "java.math.BigInteger"  
  class j-securernd  as "java.security.SecureRandom"  
  .  
working-storage section.  
  
01  params object reference http-params.  
01  http  object reference http-client
```

The classes *j-bigint* and *j-securernd* are used to create a secure random number whose purpose will be explained later.

According to Google's documentation "Before your application can use Google's OAuth 2.0 authentication system for user login, you must set up a project in the Google Developers Console (<https://console.developers.google.com/>) to obtain OAuth 2.0 credentials, set a redirect URI, and (optionally) customize the branding information that your users see on the user-consent screen. You can also use the Developers Console to create a service account, enable billing, set up filtering, and do other tasks. For more details, see the Google Developers Console Help (<https://developers.google.com/console/help/console>)"

There you get a client ID and a client secret that will be necessary in the authentication process.

Let's say that the URL of our program is <http://picosoft.it/ismobile3/OpenIDConnect> then the WORKING-STORAGE SECTION will contain:

```
78  client-id value "<client-id-by-Google>".  
78  clsc value "<client-secret-by-Google>".  
78  redir value "http://veryant.com/oauth/GOOGLEConnect".  
78  realm value "http://veryant.com/oauth".  
01  state pic x any length.
```

The login process can be divided in three stages:

- Request the authentication from Google through a redirection;
- Get the authentication data in order to be able to query Google APIs;
- Get the data about the logged user.

The program will be called two times: the first time by the user in order to start the authentication process, the second time by a Google redirection.

The first phase is simply a redirection in which you must specify what URL must be called back.

You must protect the security of your users by preventing request forgery attacks. In order to be sure that this callback is performed by the URL you actually called, a random id (*state token*) must be supplied. According to Google documentation: *"One good choice for a state token is a string of 30 or so characters constructed using a high-quality random-number generator"*. These tokens are often referred to as cross-site request forgery ([CSRF](#)) tokens.

You can create this secure random id using the classes *j-securernd* and *j-bigint* as in following code:

```
set state=j-bigint:>new(130 j-securernd:>new) :>toString(32) .
```

The code for redirection then will be:

```
phase-1-redirection.  
set state to  
    j-bigint:>new(130 j-securernd:>new) :>toString(32) .  
set params = http-params:>new  
    :>add ("client_id" client-id)  
    :>add ("response_type" "code")  
    :>add ("scope" "openid email")  
    :>add ("redirect_uri" redir)  
    :>add ("state" state)  
    :>add ("openid.realm" realm)  
comm-area:>redirect ("https://accounts.google.com/o/oauth2/auth" params) .
```

Note that the SCOPE parameter has the value "openid email": if you do not include "email" then the logger will not share his email address with your application.

The second phase begins when the same application is called back by Google, as specified by the *redir* variable. The program can easily tell if it is the first run or the second by the setting of the variables *state* and *http-state*: the former is set by phase 1 while the latter will be passed back by Google in the redirection of the login. So the initial part of the program could be the following:

```
linkage section.
01 comm-area object reference web-area.
procedure division using comm-area.
main.

    accept client-id from environment "client_id_by_google"
    accept clsc   from environment "client_secret_by_google".

    accept redir from environment "realdir".
    accept realm from environment "realm".

    if user-email = space
        perform do-auth
    else
        perform run-first-program
    end-if.
    goback.

do-auth.
    initialize http-response.
    comm-area:>accept(http-response).
    if http-state = space
        perform phase-1-redirection
    else
        if http-state = state
            perform phase-2-get-auth-token
            perform phase-3-get-info
            perform set-first-program
            perform run-first-program
        else
            comm-area:>displayError(403 "Forged state!")
        end-if
    end-if.
```

The parameters received back by Google are described in the following variable:

```
01 http-response identified by "_".
03 identified by "state".
05 http-state pic x any length.
03 identified by "code".
05 http-code pic x any length.
```

The parameter `code` (stored in `http-code`) is the one you need in order to get the authorization to query the Google APIs, along with your client ID and client secret. The source code of the second phase could be the following:

```
phase-2-get-auth-token.  
  set http = http-client:>new  
  set params = http-params:>new  
    :>add ("code" http-code)  
    :>add ("client_id" client-id)  
    :>add ("client_secret" clsc)  
    :>add ("redirect_uri" redir)  
    :>add ("grant_type" "authorization_code")  
  try  
    http:>doPost (  
      "https://accounts.google.com/o/oauth2/token"  
      params)  
    http:>getResponseCode (response-code)  
    if response-code = 200  
      http:>getResponseJSON (google-auth)  
    else  
      comm-area:>displayError(response-code "")  
      goback  
    end-if  
  catch exception  
    comm-area:>displayError(500 exception-object:>toString)  
    goback  
  end-try.
```

If the request is successful, the program will receive a JSON payload, containing two strings of characters called "access_token" and "token_type" that allow you to call anything among the Google APIs. This is the isCOBOL description of the JSON:

```
01 google-auth identified by "_".  
03 identified by "access_token".  
05 access-token pic x any length.  
03 identified by "token_type".  
05 token-type pic x any length.  
03 identified by "expires_in".  
05 expires-in pic 9(9).  
03 identified by "id_token".  
05 id-token pic x any length.
```

You still don't have any information about the person who logged in, so you need to get some basic information.

In the third phase you may choose to call the API "userinfo": this API returns a JSON payload whose data are described in the following variable:

```
01 user-info identified by "_".
   03 identified by "id".
       05 user-id pic x any length.
   03 identified by "email".
       05 user-email pic x any length.
   03 identified by "verified_email".
       05 user-verified-email pic x any length.
   03 identified by "name".
       05 user-name pic x any length.
   03 identified by "given_name".
       05 user-given-name pic x any length.
   03 identified by "family_name".
       05 user-family-name pic x any length.
   03 identified by "link".
       05 user-link pic x any length.
   03 identified by "picture".
       05 user-picture pic x any length.
   03 identified by "gender".
       05 user-gender pic x any length.
```

In order to query the Google APIs you need to put an authorization property in the header of each request: the property key will be "Authorization" while the property value will be the concatenation of the "token_type" plus the "access_token" separated by a space character. The source code could be the following:

```
phase-3-get-info.
  string token-type " " access-token into authorization
  try
    http:>setHeaderProperty ("Authorization" authorization)
    http:>doGet (
      "https://www.googleapis.com/oauth2/v2/userinfo")
    http:>getResponseCode (response-code)
    if response-code = 200
      http:>getResponseJSON (user-info)
    else
      comm-area:>displayError(response-code "")
      goback
    end-if
  catch exception
    comm-area:>displayError(500 exception-object:>toString)
    goback
  end-try.
```

If the call is successful, then the only thing to do is to start the next program, i.e. the first program in the application, for example:

```
set-first-program.  
  set environment "openid.email" to user-email.  
  accept data-dir from environment "file.prefix"  
  string data-dir "/" user-email into data-dir  
  
  call "c$makedir" using data-dir  
  set environment "file.prefix" to data-dir.  
  
run-first-program.  
  comm-area:>redirect ("_index.html").
```

For Complete examples of Facebook and Google authentications see the installed samples under `sample\eis\other\oauth`

Twitter Authentication

If you need to implement a program in order to access some Twitter's APIs using the application-only authentication, the following will explain how to do it. Also the example shows how to read some Twitts once connected. You can find Twitter's documentation at the address: <https://dev.twitter.com/docs/auth/application-only-auth>.

In order to use this kind of authentication you need to have a configured application on Twitter to get a "Consumer Key" (or "API Key") and a "Consumer secret" (or "API Secret").

These two strings are basically equivalent to a login name and a password to be used in an HTTP Basic Authentication.

Your COBOL program will define at least 2 classes: the class for doing an HTTP connection and the class for passing parameters in the HTTP requests, e.g.:

```
CONFIGURATION SECTION.  
REPOSITORY.  
    class http-client as "com.iscobol.rts.HTTPClient"  
    class http-params as "com.iscobol.rts.HTTPData.Params"  
    .  
  
WORKING-STORAGE SECTION.  
77 http object reference http-client.  
77 parms object reference http-params.
```

So the first HTTP request will be a typical POST request using the Basic authentication and supplying the parameter "grant_type" whose value will be "client_credentials".

```
set parms = http-params:>new  
    :>add ("grant_type", "client_credentials")  
  
set http = http-client:>new.  
http:>setAuth ("<Consumer-key-by-Twitter>"  
    "<Consumer-secret-by-Twitter>").  
try  
    http:>doPost (  
        "https://api.twitter.com/oauth2/token" parms)  
    http:>getResponseCode (response-code)
```

The response to this request will be a JSON-encoded payload: if the response code is different from 200 (OK), the JSON payload will be something like the following:

```
{"errors": [  
    {"label": "authenticity_token_error", "code": 99, "message":  
        "Unable to verify your          credentials"}}]
```

while if the response will be 200 the JSON payload will be something like this:

```
{"token_type":"bearer","access_token":  
"AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA2FAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA3DAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA  
AAAAAA"}
```

In order to get the data from this payload you can define the following structure in isCOBOL:

```
01 twitter-auth identified by "".  
03 identified by "token_type".  
05 token-type pic x any length.  
03 identified by "access_token".  
05 access-token pic x any length.
```

So you can get the two strings with something like:

```
if response-code = 200  
  http:>getResponseJSON (twitter-auth)
```

According to the official documentation, you must verify that the token type is "bearer" and then you can use the access token to call the APIs you need, allowed by this authentication method.

For example, you can implement the "user_timeline" API: in order to do this, we need to use the access token as "bearer" instead of the login/password used previously. The new method **setAuth (ICobolVar a)** of HTTPClient do exactly this. You can also pass all the supported parameters. See

https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline for the full documentation. E.g.:

```
if token-type = "bearer"  
  http:>setAuth (access-token)  
  set parms = http-params:>new  
    :>add ("count", "20")  
    :>add ("screen_name", "VeryantCOBOL");;  
  http:>doGet ("https://api.twitter.com/1.1"-  
    "/statuses/user_timeline.json" parms)
```

In this case you perform the GET request according to the official documentation. This request will return two different JSON payloads depending on the success of the call, but, differently from what happened in the previous API, it seems that the response code is 200 in any case. This means that you cannot know which isCOBOL structure you must use in order to get the data from the payload. Fortunately the isCOBOL JSON parser doesn't need a complete structure, it tries to fill the supplied structure with the JSON payload even if some members are missing, as long as the payload fits the structure.

The two formats returned by the above API are very different: when there is an error the format is very similar to the one already seen above when the authorization fails. If the operation return successfully, however, the payload will be an array of objects, whose length depends on the "count" parameter, each one including about 100 fields (see https://dev.twitter.com/docs/api/1.1/get/statuses/user_timeline for a complete description).

In our example we are interested only in few fields, so we have defined a structure like the following:

```
01 twitter identified by "root".
  03 identified by "errors" occurs dynamic capacity err-cnt.
    05 identified by "message".
      07 error-mesg pic x any length.
    05 identified by "code".
      07 error-code pic x any length.
  03 array identified by "element" occurs dynamic
      capacity cnt.
    05 identified by "text".
      07 twittext pic x any length.
    05 identified by "user".
      07 identified by "screen_name".
        09 screen-name pic x any length.
```

The first 03 item ("errors") is useful only when the API returns an error while the second 03 item ("element") is the data we need for our application.

This is the full program:

```
PROGRAM-ID. tweet.

CONFIGURATION SECTION.
REPOSITORY.
  class http-client as "com.iscobol.rts.HTTPClient"
  class http-params as "com.iscobol.rts.HTTPData.Params"
.

WORKING-STORAGE SECTION.
77 http object reference http-client.
77 parms object reference http-params.
77 i int.
77 some-text pic x any length.
77 response-code pic 999.

77 api-key    pic x any length.
77 api-secret pic x any length.

01 twitter-auth identified by "".
  03 identified by "token_type".
    05 token-type pic x any length.
  03 identified by "access_token".
```

```

    05 access-token pic x any length.

01 twitter identified by "root".
03 identified by "errors" occurs dynamic capacity err-cnt.
    05 identified by "message".
        07 error-mesg pic x any length.
    05 identified by "code".
        07 error-code pic x any length.
03 array identified by "element" occurs dynamic capacity cnt.
    05 identified by "text".
        07 twittext pic x any length.
    05 identified by "user".
        07 identified by "screen_name".
            09 screen-name pic x any length.

```

PROCEDURE DIVISION.

MAIN.

```

accept api-key from environment "api_key"
accept api-secret from environment "api_secret"

set parms = http-params:>new
    :>add ("grant_type", "client_credentials")

set http = http-client:>new.
http:>setAuth (api-key api-secret)
try
    http:>doPost (
        "https://api.twitter.com/oauth2/token" parms)
    http:>getResponseCode (response-code)
    if response-code = 200
        http:>getResponseJSON (twitter-auth)
        if token-type = "bearer"
            http:>setAuth (access-token)
            set parms = http-params:>new
                :>add ("count", "20")
                :>add ("screen_name", "VeryantCOBOL");;
            http:>doGet ("https://api.twitter.com/1.1"-
                "/statuses/user_timeline.json" parms)
            if response-code = 200
                display "Connection OK Response code="
                    response-code;;
                http:>getResponseJSON (twitter)
                perform show-results
            else
                display "Response code=" response-code;;
                http:>getResponsePlain (some-text)
                display some-text
                goback
            end-if
        else
            display "wrong token-type=" token-type
        end-if
    else
        display "Connection problem. Response code="
            response-code;;
        http:>getResponsePlain (some-text)
        display some-text
        goback
    end-if
catch exception

```

```
        display exception-object:>toString
        goback
    end-try.
    goback.

show-results.
display "Total Number of errors [" err-cnt "]"
if err-cnt > 0
    perform varying i from 1 by 1 until i > err-cnt
        display "code=" error-code(i) ": " error-mesg (i)
    end-perform
else
    display "Total number of Tweets [" cnt "]"
    perform varying i from 1 by 1 until i > cnt
        display "Tweet " i
        display "@" screen_name(i) ": " twittext (i)
    end-perform
end-if.
```

where "api_key" and "api_secret" are the "Consumer Key" (or "API Key") and a "Consumer secret" (or "API Secret") are retrieved from the configuration file.

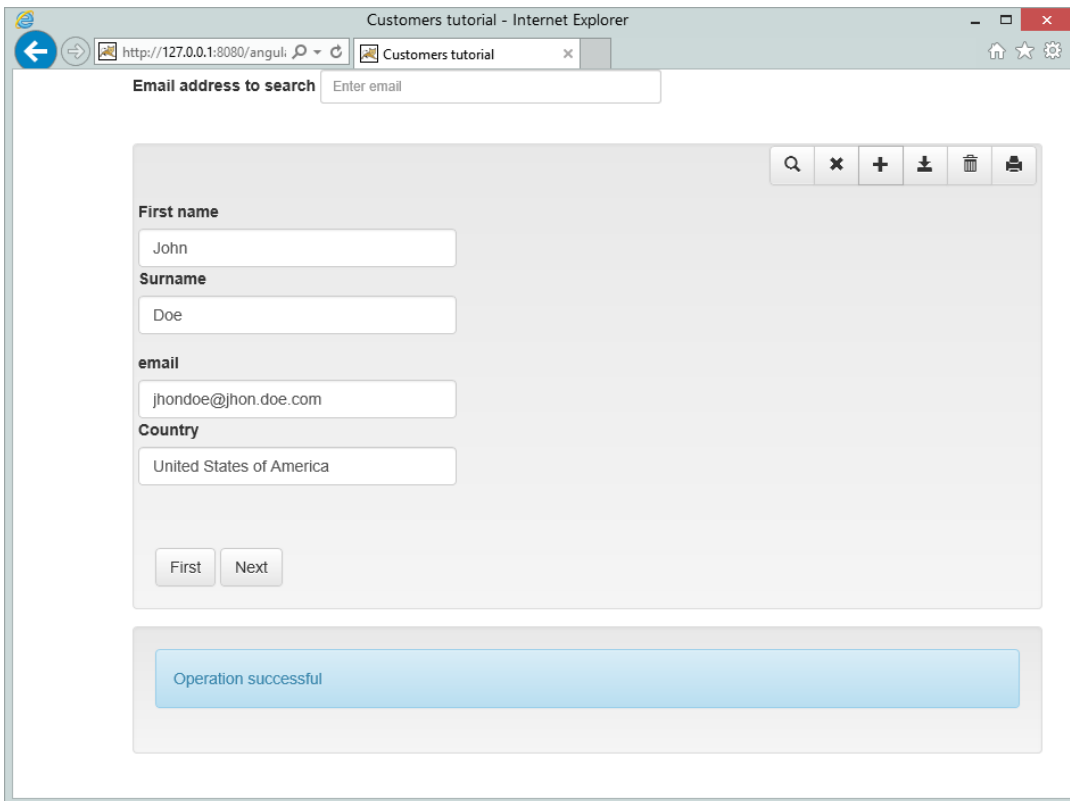
HTML5/CSS3 JS and JSON

With isCOBOL EIS taking advantage of COBOL REST producer and JSON COBOL integration, it is possible to write a Rich GUI Client Desktop application based on HTML5 and CSS3.

The client javascript library we recommend to work with is called **AngularJS** (<http://angularjs.org/>), and is developed and supported by Google.

This library, among other things, makes it easy to bind the data model coming from isCOBOL programs to the web page. An angular application is built on views and controllers. Each view (a page or part of a page) is handled by a controller, which fetches data from the isCOBOL servlet and binds it to the view's components.

The example page is described below and you find all sources on [sample/eis/other/angularjs](#).



First of all we need to include all relevant javascript and CSS files in the head of HTML page:

```
<link href="css/bootstrap.min.css" rel="stylesheet" media="screen"/>
<link href="css/bootstrap-theme.min.css" rel="stylesheet" media="screen"/>
<link href="css/customers.css" rel="stylesheet" media="screen"/>
<script type="text/javascript" src="js/jquery.js"></script>
<script type="text/javascript" src="js/angular.min.js"></script>
<script type="text/javascript" src="js/bootstrap.min.js"></script>
<script type="text/javascript" src="js/app.js"></script>
```

Then we should indicate the tag *ng-app* in the `<html>` declaration in order to load the library and process all directives in the page:

```
<html ng-app="appTutorial">
```

All code to manage HTML application is contained in the file `app.js`.

Let's examine it:

- It contains the application declaration (`angular.module('appTutorial', []);`)
- It defines a controller *CustomersCtrl* which will handle a customer page

The controller is injected with a *\$scope* variable, which represents the current instance of the controller itself, and *\$http*, which is an object provided by the AngularJS library that supports http requests to servers.

We define our model using `$scope.customer={}`, which creates an empty JSON object that will be filled by the isCobol program:

```
$scope.customer={};
```

Also, the controller defines methods to handle the buttons placed in the form, used for data navigation and processing. For example, the *getNextCustomer* method calls a COBOL entry point called AWEBX-NEXT that fetches the next customer in the dataset:

```
$scope.getNextCustomer = function() {
    $http.get("servlet/isCobol(AWEBX_NEXT)")
        .then(function (response) {
            if (response.data._comm_buffer._status=="OK")
                $scope.customer = response.data._comm_buffer;
        })
}
```

Next, we need to bind the page, or a section of a page, to a controller. In this sample we bind the *CustomerCtrl* (Customer Controller) to a div, this means that each control inside the div will have access to the model and methods defined in the controller. The binding is done using the directive

```
<div class="container" ng-controller="CustomersCtrl">
```

inside the `<div...>` tag right after the body.

Inside the div we define an HTML form, which will be handled by the CustomerCtrl as well. This is done by specifying the directive `ng-action="performSearch()"` inside the form. Form submission will trigger the PerformSearch method of the controller:

```
<form ng-submit="performSearch()" class="form-inline" role="form">
```

Notice how each button in the form is bound to a method in the controller, meaning that the click will be handled by the method specified in the `ng-click` directive. Each INPUT tag in the form is bound to one of the data model defined in the controller. In this tutorial we only have one model: customer.

The structure of this model is defined in the AWEBX.cbl COBOL source code. Each time AWEBX.cbl is executed it returns the "response" record, which contains status about the performed operation and a customer record:

```
01 comm-buffer identified by "_comm_buffer".
03 filler identified by "_status".
05 response-status pic x(2).
03 filler identified by "_message".
05 response-message pic x any length.
03 filler identified by "name".
05 json-name pic x any length.
03 filler identified by "surname".
05 json-surname pic x any length.
03 filler identified by "email".
05 json-email pic x any length.
03 filler identified by "country".
05 json-country pic x any length.
```

Let's examine how the processing of a web request is done in an Angular controller:

Take a look at the `getNextCustomer` method:

```
$scope.getNextCustomer = function() {
    $http.get("servlet/isCobol(AWEBX_NEXT)")
        .then(function(response) {
            if (response.data._comm_buffer._status=="OK")
                $scope.customer = response.data._comm_buffer;
        })
}
```

It calls the isCOBOL program, using the entry point AWEBX_NEXT. This call is asynchronous, meaning that the javascript code will continue executing while the http object is fetching the data.

When the server returns with data (or an error), the `.then()` method will be called.

The `.then` method expects as a parameter a function which receives a response object as its own parameter. The `response.data` field contains the response model defined in the `AWEBX.cbl` file.

The function needs to check if the fetch operation was successful:

```
if (response.data._comm_buffer._status=="OK")
```

and, if so, it will extract the customer model and make it available in the controller:

```
$scope.customer = response.data._comm_buffer;
```

This will automatically display the model data in the input tags of the form. Each input has a `ng-model` directive, which holds the field that will be bound to the edit field :

```
<input type="text" class="form-control"
      ng-model="customer.name"
      id="edFirstname" style="width:280px" />
```

As the user modifies the content of the input field, the model in the controller is automatically updated.

So, all we need to do in order to save the changes is to post the model to the isCOBOL program.

This is done in the `saveCustomer` (or `newCustomer`) method of the controller. All we need to do is call the isCOBOL program, using the right entry point, `AWEBX_UPDATE` (`AWEBX_INSERT`), and pass it the customer model :

```
$scope.saveCustomer = function() {
    callServerWithJson("AWEBX_UPDATE",
    $scope.customer, $scope.onSuccess, $scope.onError);
}
```

The `callServerWithJson` utility method accepts the entry point to call, a model to pass to the isCOBOL program, and 2 callbacks, that specify the function to execute if the http request is successful *onSuccess*, or if it fails *onError*.

Notice that the *onSuccess* will be called even if the isCOBOL program generates an error (duplicated key, record locked, and so on). This is because the HTTP request was carried out successfully, but a logical program error occurred. So the *onSuccess* method needs to check the response object and handle it appropriately.

The *onError* callback will be invoked only if the http request fails (network error, server error,...)

isCOBOL EIS (Enterprise Information System), COBOL Servlet option (OOP)

One of the initial purposes of the Java language was to enable programmers to make Web pages more interactive by embedding programs called applets. When a browser loads a Web page containing an applet, the browser downloads the applet byte code and executes it on the client machine. However, because of client compatibility, bandwidth, security and other issues, businesses needed an alternative solution where Web pages could be made to interact with server-side instead of client-side Java programs.

Server-side Java programming solves problems associated with applets. A servlet can be thought of as a server-side applet. However, when the code is executed on the server-side, there are no issues with browser compatibility or download times. The servlet byte code runs entirely on the server and only sends information to the client in a form that the client can understand.

Similar to a CGI program, a servlet takes requests from a client such as a Web browser, accesses data, applies business logic, and returns the results.

The servlet is loaded and executed by the Web server, and the client communicates with the servlet through the Web server using HTTP requests. This means that if your Web server is behind a firewall, your servlet is secure.

Servlet technology was developed to improve upon and replace CGI programs. Servlet technology is superior to CGI but uses the same HTML code. So you can switch from CGI programs to servlets on the back-end without having to change the programming on the front-end. Servlets use the CGI protocol.

In addition to Java technology's platform independence and promise of write once, run anywhere, servlets have other advantages over CGI programs:

- Servlets are persistent. They are loaded only once by the Web server and can maintain services such as database connections between requests.
- Servlets are fast. They need to be loaded only once by the Web server. They handle concurrent requests on multiple threads rather than in multiple processes. Thus, applications with servlets perform better and are more scalable than the same applications using CGI programs.
- Servlets are platform and Web server independent.
- Servlets can be used with a variety of clients, not just Web browsers.

- Servlets can be used with a variety of client-side and server-side Web programming techniques and languages.

The isCOBOL EIS introduces a new way to develop COBOL programs that acts like java servlet using **HTTPHandler** class functionality.

One of the most remarkable differences between COBOL servlets and CGI programs is that Web servers automatically maintain user session state for servlets. This means that the COBOL servlet can store user-session specific information in a user session object and retrieve that information on a subsequent call.

The isCOBOL EIS Framework uses this feature to associate the user session with a COBOL thread context. This makes sure that the same instances of COBOL programs get used each time they are called during a particular user session. In other words, COBOL programs called during a particular user session retain their file states and working-storage data between requests from that user session. If desired, the programmer can cancel the program at any time with the CANCEL statement. In fact, at first it will be necessary to cancel old CGI programs because they were written to assume that they have been cancelled between calls. Later, the CANCEL statement can be removed as the old CGI programs are updated to make use of the Stateful nature of COBOL servlets.

COBOL Servlet Programming

Following you will find an explanation about how to develop a simple COBOL servlet that builds an HTML page using a header.htm page and a footer.htm page, filling them with the correct message and sending a text string between them, the string is "Hello world from isCOBOL!".

The Web Servlet container used for this example is Tomcat 7.

This program needs to take the following steps:

- Create a folder called *doctest* with the following structure:

```
doctest/  
    WEB-INF/  
        classes  
        lib
```

- Create a file called web.xml in doctest/WEB-INF folder with the following content:

```
<?xml version="1.0" encoding="UTF-8" ?>  
<web-app xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance  
  xmlns=http://java.sun.com/xml/ns/javaee  
  xmlns:web=http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd  
  xsi:schemaLocation=http://java.sun.com/xml/ns/javaee  
http://java.sun.com/xml/ns/javaee/web-app\_2\_5.xsd id="WebApp_ID"  
  version="2.5">  
  <display-name>isCOBOL EIS</display-name>  
  <welcome-file-list>  
    <welcome-file>Hello.htm</welcome-file>  
  </welcome-file-list>  
  <filter>  
    <filter-name>isCOBOL filter</filter-name>  
    <filter-class>com.iscobol.web.IscobolFilter</filter-class>  
  </filter>  
  <filter-mapping>  
    <filter-name>isCOBOL filter</filter-name>  
    <url-pattern>/servlet/*</url-pattern>  
  </filter-mapping>  
  <servlet>  
    <servlet-name>isCobol</servlet-name>  
    <servlet-class>com.iscobol.web.IscobolServletCall</servlet-class>  
  </servlet>  
  <servlet-mapping>  
    <servlet-name>isCobol</servlet-name>  
    <url-pattern>/servlet/*</url-pattern>  
  </servlet-mapping>  
  <listener>  
    <listener-class>com.iscobol.web.IscobolSessionListener</listener-class>  
  </listener>  
</web-app>
```

- Create a *Hello.htm* web form to call a COBOL servlet called HELLO:

```
<HTML><HEAD><TITLE>Doc isCOBOL Example</TITLE></HEAD>
<BODY>
<H2>Doc isCOBOL Example.</H2>
<H3>This example shows how easily you can compose an HTML page with an isCOBOL
program running on the web server. The HTML page is composed by two parts an
header and a footer. In every part of the HTML page, the isCOBOL program moves
a message and between the two sends the text: "Hello world from isCOBOL".</H3>
<HR size="2">
<FORM method="post" action="servlet/isCobol (HELLO) ">
  <p><input type="submit" value="Invoke isCOBOL HELLO program" /></p>
</FORM>
```

Note that in POST method of HTML form there is the call of the COBOL Servlet called *HELLO*.

- Create a *Header.htm* as follow:

```
<HTML>
<HEAD><TITLE>CGI Header</TITLE></HEAD>
<BODY>
<CENTER>
<H1>This is the Header HTM form of this isCOBOL example</H1>
<H2>This is the message sent by the isCOBOL program: %%opening-message%%</H2>
<HR>
```

Note that this form displays the top of the HTML page that the program *HELLO.cbl* will build; as we can see, the `<HTML>`, `<BODY>` and `<CENTER>` tags are not closed, and there is the string `%%opening-message%%` that will be managed and replaced by the COBOL servlet program

- Create a *Footer.htm* web form as follow:

```
</CENTER>
<BR>
<HR>
This is the footer HTML page of this isCOBOL example.
<H2>This is the message sent by the program: %%closing-message%%</H2>
</BODY></HTML>
```

Note that this form displays the bottom of the HTML page that the program *HELLO.cbl* will build. Here the tags `<HTML>`, `<BODY>` and `<CENTER>` are closed and there is the string `%%closing-message%%` that will be managed and replaced by the COBOL servlet program.

- Create a *HELLO.cbl* COBOL Servlet program as follows:

```

PROGRAM-ID. HELLO initial.
CONFIGURATION SECTION.
REPOSITORY.
class web-area as "com.iscobol.rts.HTTPHandler"
.
WORKING-STORAGE SECTION.
01 hello-buffer pic x(40) value "Hello World from isCOBOL!".
01 rc pic 9.
01 html-header-form identified by "Header".
   05 identified by "opening-message".
   10 opening-message pic x(40).
01 html-footer-form identified by "Footer".
   05 identified by "closing-message".
   10 closing-message pic x(40).

LINKAGE SECTION.
01 comm-area object reference web-area.
PROCEDURE DIVISION using comm-area.
MAIN-LOGIC.
   move "This is the header" to opening-message
   set rc = comm-area:>processHtmlFile (html-header-form).
   comm-area:>displayText (hello-buffer).
   move "Bye Bye by isCOBOL" to closing-message
   set rc = comm-area:>processHtmlFile (html-footer-form).
   goback.

```

Note that the COBOL servlet does the following steps:

- Move the value "This is the header" to the variable *opening-message* of the structure prepared for *Header.htm*
- Add to the HTML page source (that currently is empty) the *Header.htm* form replacing the string `%%opening-message%%` by the *opening-message* variable value
- Add to the HTML page the text "Hello world from isCOBOL!"
- Move the value "Bye Bye by isCOBOL" to the variable *closing-message* of the structure prepared for *Footer.htm*
- Add to the HTML page source the *Footer.htm* form replacing the string `%%closing-message%%` by the *closing-message* variable value

at the exit of the program, the page HTML will be sent to the Web Server.

- Compile *HELLO.cbl* without particular option and copy *HELLO.class* under *doctest/WEB-IF/classes* folder
- Create a *iscobol.properties* file under *doctest/WEB-IF/classes* folder with a property to inform isCOBOL EIS framework of the path of all HTML useful files:
`iscobol.http.html_template_prefix=webapps/doctest`

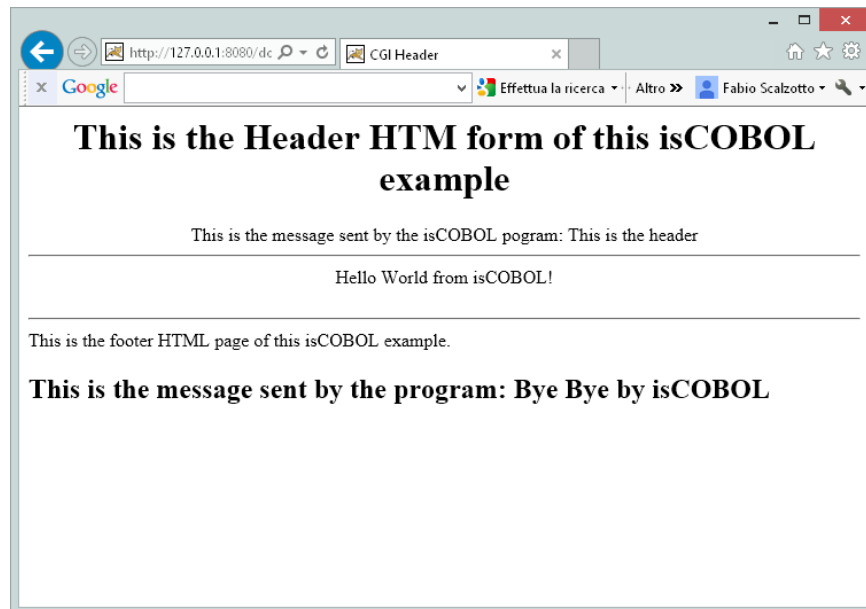
- Create a war file to be deployed in Tomcat called *doctest.war* that includes all files of doctest folder. It can be done with the following command:

```
jar -cfv doctest.war *
```

Once doctest.war file is deployed correctly in Tomcat servlet container, we can try it using *http://127.0.0.1:8080/doctest*, assuming to have Tomcat running on localhost using default 8080 port :



By pressing the “Invoke isCOBOL Hello program” button, the result is:



COBOL Servlet Programming with AJAX and XML

AJAX (Asynchronous JavaScript and XML) is a group of interrelated Web development techniques used on the client side to create asynchronous Web Applications. With Ajax, Web applications can send data to, and retrieve data from, a server asynchronously (in the background) without interfering with the display and behavior of the existing page.

Extensible Markup Language (XML) is a text format derived from Standard Generalized Markup Language (SGML). Compared to SGML, XML is simple. HyperText Markup Language (HTML), by comparison, is even simpler. Even so, a good reference book on HTML is an inch thick. This is because the formatting and structuring of documents is a complicated business.

Most of the excitement about XML is related to a new role as an interchangeable data serialization format. XML provides two enormous advantages as a data representation language:

- It is text-based
- It is position-independent

The scope of this paragraph is to show how to develop a simple web application that uses XML stream to communicate data from COBOL servlet to a Web form.

The following example called HELLO.cbl, is located in sample/eis/http/xml folder. The README.txt file explains how it works and how to deploy it.

This example needs to take the following steps:

- Create a HTML file called index.html that is able to establish a AJAX communication to receive a XML stream from COBOL servlet program:

In index.html there is included a Javascript code based on JQUERY to be able to call some COBOL servlet entry point making a GET request type (default) and receiving XML data stream:

```
function callServer (cobolProg) {
    var url = "servlet/isCobol(" + cobolProg + ")";
    jQuery.ajax(url, {
        success: handleSuccess,
        error: handleError
    });
    return false;
}
```

- o Load all COBOL Servlets using the following statement:

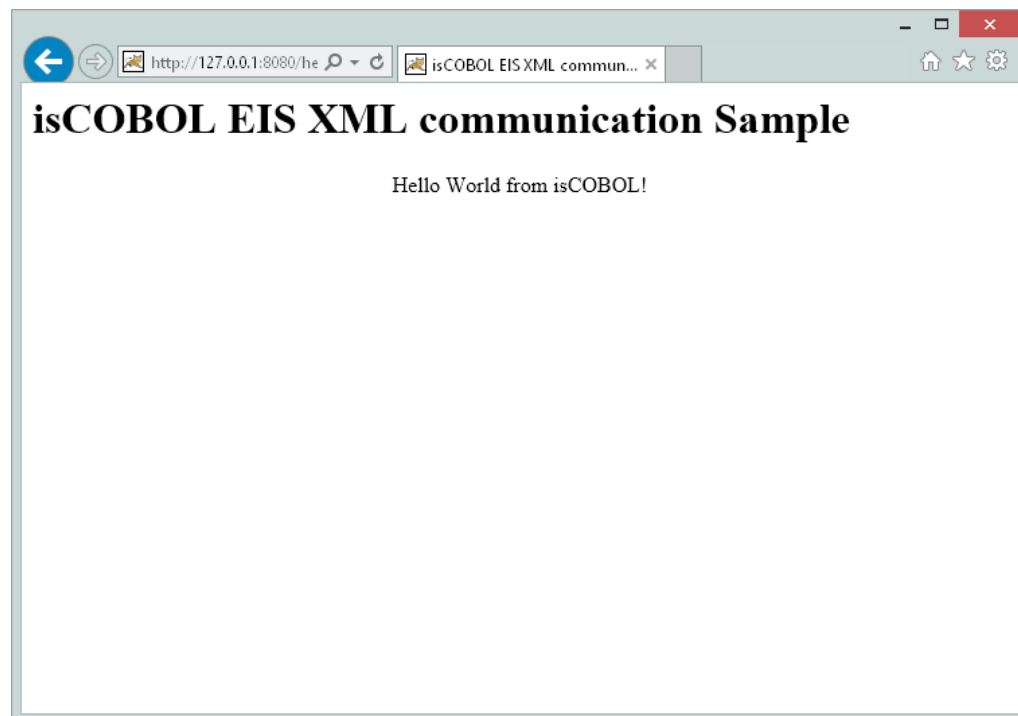
```
window.onload = callServer('HELLO');
```

Note the when HELLO COBOL Servlet is loaded the following code executed:

```
move "Hello World from isCOBOL!" to xml-hellotext.  
lnk-area:>displayXML (hello-buffer).
```

And XML stream is returned to Web form with *displayXML()* command.

Running this example the result is the following:



COBOL Servlet Programming with AJAX and JSON

JSON or JavaScript Object Notation, is an open standard format that uses text easy to understand to transmit data objects consisting of attribute-value pairs. It is used primarily to transmit data between a server and web application, as an alternative to XML.

Although originally derived from the JavaScript scripting language, JSON is an independent data format, and code for parsing and generating JSON data is readily available in a large variety of programming languages.

Here we will show how to develop a simple web application of data file management that uses JSON stream to communicate data from COBOL servlet to HTML pages.

The following example is located in sample/eis/http/json folder. The README.txt file explains how it works and how to deploy it.

This example needs to take the following steps:

- Having a HTML file that is able to establish a AJAX communication using JSON stream to a COBOL servlet program:

In awebx.htm there is included a Javascript code based on JQUERY to be able to call some COBOL servlet entry point making a GET request type (default) and receiving JSON data stream:

```
function callServer (cobolProg) {
    var url = "servlet/isCobol(" + cobolProg + ")";
    var parm = $("form").serialize();
    $.ajax(url, {
        success: handleSuccess,
        error: handleError,
        data: parm
    });
    return false;
}
```

- o Load all COBOL Servlet entry points using the following statement:

```
callServer("AWEBX"); // program initialization
```

Note the once AWEBX COBOL Servlet is loaded the INIT paragraph is executed:

```
INIT.  
  set declaratives-off to true.  
  move low-values to r-awebx-email.  
  open i-o awebxfile.  
  set declaratives-on to true.  
  if file-status > "0z" and file-status not = "41"  
    open output awebxfile  
    close awebxfile  
    open i-o awebxfile.  
  comm-area:>displayJSON (ok-page).  
  goback.
```

- o associates to every HTML button a AWEBX entry point to be executed when the button is clicked :

```
<input type="submit" value="Insert" onclick="return callServer('AWEBX_INSERT');">  
<input type="submit" value="Search" onclick="return callServer('AWEBX_SEARCH');">  
<input type="submit" value="Next" onclick="return callServer('AWEBX_NEXT');">  
<input type="submit" value="Update" onclick="return callServer('AWEBX_UPDATE');">  
<input type="submit" value="Delete" onclick="return callServer('AWEBX_DELETE');">
```

Note that the above HTML is able to call the following COBOL servlet entry-point:

```
INSERT-VALUES.  
  entry "AWEBX_INSERT" using comm-area.  
  ...  
  goback.
```

```
SEARCH-VALUES.  
  entry "AWEBX_SEARCH" using comm-area.  
  ...  
  goback.
```

```
NEXT-VALUES.  
  entry "AWEBX_NEXT" using comm-area.  
  ...  
  goback.
```

```
UPDATE-VALUES.  
  entry "AWEBX_UPDATE" using comm-area.  
  ...  
  goback.
```

- Define in HTML some fields to input data suitable for data management like name, surname, email, country etc:

```
<input type="text" name="name" size="25" placeholder="Name"/><br/>
<input type="text" name="surname" size="25" placeholder="Surname"/><br/>
<input type="text" name="email" size="25" placeholder="E-mail"/><br/>
<select name="country" placeholder="Country">
  <option value="" selected="selected" disabled="disabled">Country</option>
  <option value="us">US</option>
  <option value="it">Italy</option>
  <option value="fi">Finland</option>
  <option value="nl">The Netherlands</option>
  <option value="de">Germany</option>
  <option value="fr">France</option>
  <option value="sp">Spain</option>
  <option value="uk">United Kingdom</option>
</select><br/>
```

- On COBOL Servlet create a working storage structure that matches the field name of previous HTML. It can be done with **identified by** clause:

```
01 comm-buffer identified by "_comm_buffer".
03 filler identified by "_status".
   05 response-status pic x(2).
03 filler identified by "_message".
   05 response-message pic x any length.
03 filler identified by "name".
   05 json-name pic x any length.
03 filler identified by "surname".
   05 json-surname pic x any length.
03 filler identified by "email".
   05 json-email pic x any length.
03 filler identified by "country".
   05 json-country pic x any length.
```

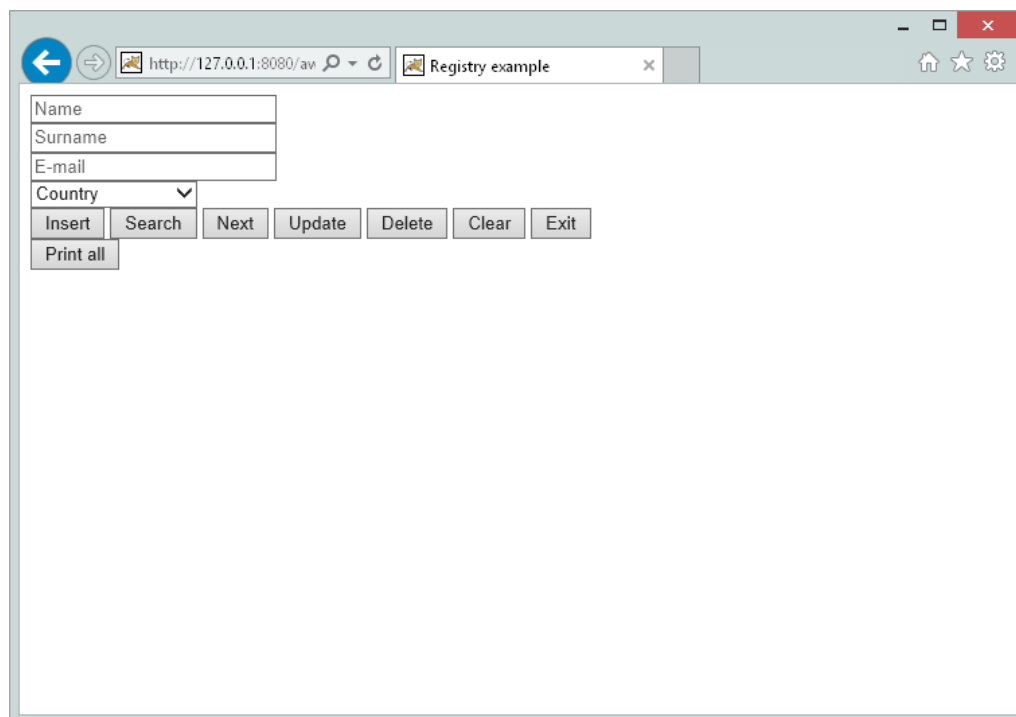
- On COBOL Servlet manage GET request by accept() answering with a JSON stream by displayJSON(). For example if "insert" is submitted the following entry point is invoked:

```
INSERT-VALUES.
  entry "AWEBX_INSERT" using comm-area.
  comm-area:>accept(comm-buffer).
  move spaces to error-status.
  perform check-values.
  if error-status = spaces
    move json-name to r-awebx-name
    move json-surname to r-awebx-surname
    move json-email to r-awebx-email
    move json-country to r-awebx-country
    write rec-awebxfile
    move "Operation successful" to ok-message;;
    comm-area:>displayJSON(ok-page)
  else
    comm-area:>displayJSON(error-page).
  goback.
```

- o In a similar way when a "next" command is submitted, the records are returning back as JSON stream with displayJSON() command:

```
NEXT-VALUES.  
entry "AWEBX_NEXT" using comm-area.  
read awebxfile next  
move r-awebx-name to json-name  
move r-awebx-surname to json-surname  
move r-awebx-email to json-email  
move r-awebx-country to json-country  
move "OK" to response-status  
move "" to response-message;;  
comm-area:>displayJSON (comm-buffer).  
goback.
```

This is the output form of awebx.htm used in previous example:



The screenshot shows a web browser window with the address bar displaying 'http://127.0.0.1:8080/aw'. The page title is 'Registry example'. The form contains the following elements:

- Text input fields for 'Name', 'Surname', and 'E-mail'.
- A dropdown menu for 'Country'.
- A row of buttons: 'Insert', 'Search', 'Next', 'Update', 'Delete', 'Clear', and 'Exit'.
- A 'Print all' button located below the row of buttons.

COBOL Servlet Programming to replace CGI COBOL programming

The scope of this paragraph is to show how to migrate older CGI COBOL program to isCOBOL Servlet taking advantage of useful features of HTTPHandler class. Usually few changes are required and most of the sources will be unchanged.

The following example is located in sample/eis/http/getpost/acucgi2is folder. The README.txt file explains how it works and how to deploy it.

This example needs to take the following steps:

- o Having COBOL Servlet invocation in POST form action to specify the name of COBOL program that acts like CGI program:

```
<FORM method="post" action="servlet/isCobol(ISOSCARS)">
```

- o Assuming to have a Web form called oscar.htm with the following controls:

```
<input type=checkbox name=y1996 value=1996> 1996  
<input type=checkbox name=y1995 value=1995> 1995  
<input type=checkbox name=y1994 value=1994> 1994  
<input type=checkbox name=y1993 value=1993> 1993  
<P>  
<input type=checkbox name=y1992 value=1992> 1992  
<input type=checkbox name=y1991 value=1991> 1991  
<input type=checkbox name=y1990 value=1990> 1990  
<input type=checkbox name=y1989 value=1989> 1989  
<P>  
<input type=checkbox name=y1988 value=1988> 1988  
<input type=checkbox name=y1987 value=1987> 1987  
<input type=checkbox name=y1986 value=1986> 1986  
<input type=checkbox name=y1985 value=1985> 1985  
<P>  
<input type="submit" value="Submit Query" >
```

Checking one or more years and pressing the 'Submit query' button, the ISOSCARS COBOL servlet program is called and it uses the HTTPHandler class to communicate with the Web form.

- o The old ACUCOBOL-GT CGI program should be changed in order to be transformed in a COBOL servlet :
 - o Include the HTTPHandler class in beginning of COBOL CGI program:

```
configuration section.  
repository.  
class web-area as "com.iscobol.rts.HTTPHandler"
```


- o In order to be able to accept the HTML page input, cgi-form is defined as follows:

```
01 cgi-form identified by "cgi-form".
   05 identified by "y1996".
       10 y1996 pic x(5).
   05 identified by "y1995".
       10 y1995 pic x(5).
   05 identified by "y1994".
       10 y1994 pic x(5).
   05 identified by "y1993".
       10 y1993 pic x(5).
   05 identified by "y1992".
       10 y1992 pic x(5).
   05 identified by "y1991".
       10 y1991 pic x(5).
   05 identified by "y1990".
       10 y1990 pic x(5).
   05 identified by "y1989".
       10 y1989 pic x(5).
   05 identified by "y1988".
       10 y1988 pic x(5).
   05 identified by "y1987".
       10 y1987 pic x(5).
   05 identified by "y1986".
       10 y1986 pic x(5).
   05 identified by "y1985".
       10 y1985 pic x(5).
```

- o Then accept() should be used to receive parameters into working storage replacing legacy ACCEPT:

```
linkage section.
01 comm-area object reference web-area.
procedure division using comm-area.

       comm-area:>accept(cgi-form).
```

- o Depending on the input field received by the accept statement, the program fills the HTML page to send. First of all it sends the header using the following statement:

```
move "CGI in action." to opening-message
set rc = comm-area:>processHtmlFile (html-header-form)
```

where html-header-form is defined as follows:

```
01 html-header-form identified by "header".
   05 identified by "opening-message".
       10 opening-message pic x(40).
```

The strings specified with the identified by clause are the name of the page and the name of the fields in it. So the header.htm page is used. The string %%opening-message%% is replaced by the opening-message variable value.

The legacy DISPLAY should be replaced with processHTMLFile() statement.

The program ISOSCARS goes ahead and adds the body.htm page after the replacement of the values of the years requested have been applied. In the same way footer.htm is added at the end. The result choosing 1994, 1992 and 1986 is the following:

CGI in action.

Oscar Winners

Your Selections			
Year	Best Movie	Best Actor	Best Actress
1994	FORREST GUMP	Tom Hanks FORREST GUMP	Jessica Lange BLUE SKY
1992	UNFORGIVEN	Al Pacino SCENT OF A WOMAN	Emma Thompson HOWARDS END
1986	PLATOON	Paul Newman THE COLOR OF MONEY	Marlin Matlin CHILDREN OF A LESSER GOD

THE END.

The information you requested was processed by the CGI program. Following the CGI standard, isCOBOL was able to send the requested data items to the appropriate templates and return the completed HTML document back to you.

Using the above approach is also possible to migrate a Micro Focus COBOL CGI program to COBOL Servlet.

Under sample/eis/http/getpost/ mfcgi2is folder you find an example of a Micro Focus Cobol CGI program rewritten to run with the HTTP option of isCOBOL EIS.

The README.txt file explains how it works and how to deploy it.

isCOBOL EIS (Enterprise Information System), Web Direct 2.0 option

With isCOBOL EIS WebDirect 2 option your organization can leverage existing COBOL syntax to develop and deploy a universally accessible, zero client, rich Internet application (RIA) using standard COBOL screen sections and existing program procedure flow. No knowledge of object-oriented programming, JavaScript, HTML, or other Web languages is required.

isCOBOL EIS Web Direct 2.0 (EIS WD2) is a Java framework for presenting a "graphical" user interface, composed of elements such as windows, dialogs, menus, text fields and buttons, inside a Web Browser. This technology uses AJAX (asynchronous JavaScript and XML) techniques and the Comet web application model. The web application is deployed as a Servlet and therefore requires a Java-enabled web server, one that implements the Java Servlet specification from Sun Microsystems

isCOBOL EIS Web Direct 2.0 takes advantages of ZK libraries, installed with the product. ZK is an event-driven, component-based framework to enable rich user interfaces for Web applications. ZK includes an AJAX[1]-based event-driven engine, a rich component set of XUL and XHTML and a markup language called ZUML (ZK User Interface Markup Language).

Technical Notes

EIS WD2 on the client side is a JavaScript application running inside a web browser. This environment has many limitations in comparison with a full GUI environment, e.g. only a few events are generated. JavaScript is a script language so its performance is not as good as compiled languages, although latest generation browsers are improving performance by the use of JIT (Just In Time) compilers.

EIS WD2 was not developed from scratch; it uses a library, ZK, that hides the JavaScript implementation and exposes a Java API. Veryant interfaces our set of GUI controls with the ones implemented in ZK. As a result, because our controls are similar to those provided by ZK, future updates will require less effort and provide more stable releases interfacing with ZK GUI controls. Alternatively, controls completely different from the ZK controls will require more development and testing time.

The client/server communication is performed through the HTTP protocol; since this protocol is very limited in functionality, a special technique (called "COMET") has been used in order to get the needed functionality. This technology is the up-to-date best

technology in this area. However its performance is not as good as native protocols. Just as an example, it uses XML protocols, so it creates bigger messages and it requires considerable computation resources for marshalling.

Installation Environment

In order to deploy and run programs using Web Direct 2.0, the following environment must be set up a servlet container like Apache Tomcat.

Veryant recommends using Apache Software Foundation Tomcat version 7 for running EIS WD2 Application.

The Apache Tomcat main page is <http://tomcat.apache.org/>

isCOBOL EIS WebDirect 2.0 is expected to work also on the following containers:

- IBM WebSphere
- BEA WebLogic
- JBoss
- Oracle OC4J and Oracle OPMN Release 3
- Liferay
- Pluto
- Jetty
- Resin

Servlet container and Web Browser Requirements

Web Direct 2.0 runs on any web server that supports Servlet 2.3+ and JVM 1.4+.

The web browser must be able to run JavaScript and support Ajax (namely the XMLHttpRequest object).

Examples are:

Internet Explorer 5+

Firefox 1+

Mozilla 1+

Safari 1.2+

any version of Google Chrome.

isCOBOL EIS WD2 Getting Started

The jar libraries must be copied in the proper directory in order to be available to the web application. If you're using Tomcat, you must copy these libraries in the "lib" folder of your web application.

Web Direct 2.0 is composed by:

- The isCOBOL Framework : **isrun.jar**
- Web Direct 2.0 classes that extends the Framework: **iswd2.jar**
- ZK libraries
- **web.xml**: also known as Deployment Descriptor. To configure servlets, listeners and an optional filter
- **zk.xml**: the configuration descriptor of ZK. This file is optional. If you need to configure ZK differently from the default, you could provide a file called zk.xml under the WEB-INF directory.

All the above files are provided by the library wd2.war, installed in \$ISCOBOL/sample/wd2. You can extract them with the jar command or you can deploy the sample application as explained in the Running the sample application section.

To extract the files with the jar command, use:

```
jar -xf wd2.war
```

isCOBOL EIS WD2 Running a sample application

Web Direct 2.0 comes with a sample web application. This chapter explains how to deploy and run the sample application.

Download Tomcat from <http://tomcat.apache.org/> and install it, if you haven't installed it yet. Start the Tomcat service.

Note: if you're running Tomcat on Unix/Linux, ensure that the working directory is the Tomcat home directory. If you start the process from another directory (e.g. the Tomcat bin directory), then relative paths in the sample will not work.

When Tomcat service is started, open a browser and navigate to:

<http://127.0.0.1:8080/>. The browser displays something like:



Select Tomcat Manager link in order to application administration pages. You will be prompted for username and password. By default Tomcat has the user "admin" with no password. You can refer to tomcat-users.xml

Using the Tomcat Web Application Manager, scroll down to the Deploy dialog and use the Browse button to select the Web Application Archive file (wd2.war)

Deploy

Deploy directory or WAR file located on server

Context Path (required):

XML Configuration file URL:

WAR or Directory URL:

WAR file to deploy

Select WAR file to upload

Diagnostics

Check to see if a web application has caused a memory leak on stop, reload or undeploy

This diagnostic check will trigger a full garbage collection. Use it with extreme caution on production systems.

Server Information

Tomcat Version	JVM Version	JVM Vendor	OS Name	OS Version	OS Architecture	Hostname	IP Address
Apache Tomcat/7.0.37	1.7.0_10-b18	Oracle Corporation	Windows 8	6.2	amd64	Luciano-Veryant	192.168.0.213

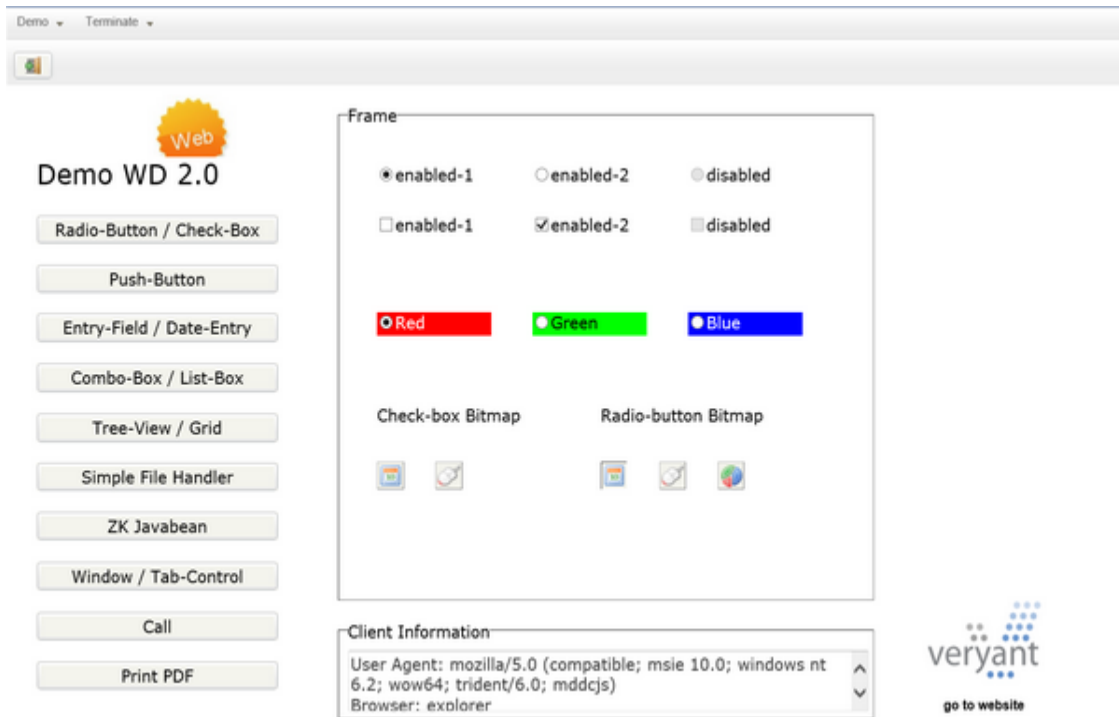
An item called "wd2" will be added to the Applications list and Tomcat will create a structure similar to this in the webapps directory:

- ▲ wd2
 - META-INF
 - pdf
 - ▲ resource
 - css
 - images
 - upload
 - ▲ WEB-INF
 - arc
 - classes
 - lib
 - programs

Edit the file iscobol.properties in classes' folder to insert valid license codes.

To run the sample, open a browser and navigate to:

http://127.0.0.1:8080/wd2



isCOBOL EIS WD2 Guidelines for writing a web application

Web Direct 2.0 allows bringing GUI COBOL programs into the web without specific modification.

Each COBOL program with a Screen Section containing graphical controls can run as web application with Web Direct 2.0.

However, not all GUI features are supported by Web Direct. If you plan to bring an existing COBOL application into the web It is strongly suggested to compile all sources with the -wd2 option. In this case the isCOBOL Compiler will alert you with warning messages if an unsupported feature is being used. You can consult the Unsupported Features chapter to see what is not allowed by Web Direct 2.0.

In order to produce a fast web application, It is strongly suggested to:

- Reduce the number of controls in the screen
- Avoid using embedded and event procedures if not necessary

It's very important to avoid using STOP RUN statement if you plan to run your programs as web application. STOP RUN causes the whole JVM to exit and it would result in the shutdown of the whole servlet container. Use GOBACK instead.

Developing a hello world application from scratch

The next chapter illustrates the steps to create a hello world application from scratch, compile it, deploy it and eventually debug it.

Writing the source

Programs for the web are standard COBOL programs. The following source code produces a screen with a button with "Hello World" inside:

```
PROGRAM-ID. HELLO.
SCREEN SECTION.
01 SCREEN1.
    03 PUSH-BUTTON
        LINE 4
        COL 4
        SIZE 15
        HEIGHT-IN-CELLS
        WIDTH-IN-CELLS
        TITLE "Hello World"
        EXCEPTION-VALUE 100
    .
PROCEDURE DIVISION.
MAIN.
    DISPLAY STANDARD GRAPHICAL WINDOW.
    DISPLAY SCREEN1.
    ACCEPT SCREEN1 ON EXCEPTION CONTINUE.
```

Compiling the source

Since we plan to debug the program after the deployment, we'll use the -d option.

The -wd2 option is also used to be sure that our program is compatible with Web Direct.

```
iscc -d -wd2 hello.cbl
```

Creating the configuration file

In order to run with Web Direct 2.0 we must instruct the program to use a specific guifactory class.

In addition, the license codes for the isCOBOL Framework and Web Direct 2.0 must be provided, so our configuration file will look like this:

```
iscobol.guifactory.class=com.iscobol.gui.client.zk.GuiFactoryImpl
iscobol.license.2014=XXXXXXXXXXXXXXXXXXXXXXXXXXXX
iscobol.wd2.license.2014=XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

The configuration file will be placed between program classes in the webapp directories. However, the configuration is also loaded from \etc directory and from the user home directory depending on the drive where Tomcat was started and on the user that owns its process.

Deploying in Tomcat

The easiest way to deploy a new web app is to:

Deploy the WD2 sample program as explained in Running the sample application on Tomcat chapter

- Deploy the WD2 sample program as explained in Running the sample application on Tomcat chapter
- Make a copy of the tomcat/webapps/wd2 folder and rename the copy to the name of your choice (i.e. 'test')
- Add your class files to one of the following:
 - the WEB-INF/classes folder
 - a jar file placed in the WEB-INF/lib folder
 - one of the folders listed in iscobol.code_prefix configuration property
- Add your properties to the WEB-INF/classes/iscobol.properties file
- Restart Tomcat

Running the application

From Web browser considering our program that is called HELLO, you will use the following URL: <http://127.0.0.1:8080/test/zk/lsMainZK?PROGRAM=HELLO> to have:

Hello World

Debugging

In order to debug the web application

- Programs must be compiled with -d option
- The following entry must appear in the configuration

```
iscobol.rundebbug=2
```

The Remote Debugger feature is used

When you connect to the page of your application you will see a blank page. It means that the web application is waiting for the Debugger to connect. Launch the following command to use the Debugger:

```
iscrun -J-Discobol.debug.code_prefix=sourcePath -d -r serverIp
```

Where:

- *sourcePath* is the list of paths where program source code and copyfiles can be found
- *serverIp* is the ip (or name) of the web-server where the web application is running, in our case: 127.0.0.1

If everything has been done correctly, you should see the web page show up while you debug the DISPLAY statements.

Using Native Libraries inside isCOBOL EIS WD2

Usually c-treeRTG and other file handlers provide a file connector solution. When a file connector is available, it's preferable to use it instead of using native libraries. In order to use the file connector you just set the `iscobol.file.index` and `iscobol.file.connector.program(.connector_name)` properties to proper values in the `iscobol.properties` file installed in your webapp.

If a file connector is not available or you have to use other native libraries for features not related to file handling, proceed as follows:

- If the servlet container (Tomcat) is running on Windows, the folder containing the native library must appear in the PATH (System PATH setting, not User PATH). Alternatively, you can copy the necessary native libraries into the Tomcat bin folder.
- If you're working on UNIX/Linux, instead, ensure that the directory containing the native library is listed in the library path (e.g. `LD_LIBRARY_PATH`, `LIBPATH`, `SHLIB_PATH`, etc.)

For example, in a typical configuration `/etc/tomcat7/tomcat7.conf` sources `/usr/share/tomcat7/bin/setenv.sh` which is the appropriate place to set global `CLASSPATH` and `LD_LIBRARY_PATH` for Tomcat. In some cases, you can also set variables in `$HOME/.tomcatrc`.

If you're using a container different than Tomcat, consult the documentation for the specific product.

How to receive parameters in EIS WD2

Programs can receive parameters from the URL.

Parameters must be added at the end of the URL using the *syntax* `&PARAM_NAME=Value` and they're intercepted by the COBOL program as chaining parameters.

The following COBOL program, for example, expects 2 parameters, p1 and p2:

```
PROGRAM-ID. prog.  
WORKING-STORAGE SECTION.  
77 p1 pic x(10).  
77 p2 pic x(10).  
PROCEDURE DIVISION chaining p1 p2.  
main.  
    display message "p1=" p1.  
    display message "p2=" p2.  
    Goback.
```

The parameters will be passed using a URL like:

<http://127.0.0.1:8080/wd2/zk/lsMainZK?PROGRAM=PROG&P1=AAA&P2=BBB>

How to Handle Program Exit

By default, when the program terminates due to GOBACK statement, the last screen remains in the web-browser, but is no longer active. This may result in the impression that the program hanged, while it was just terminated.

The proper way to handle the program exit, is by redirecting the browser to a different web page, that may be the page from which the application was launched or the home page of your website or whatever else.

This objective is achieved through JavaScript.

In order to make WebDirect 2.0 execute JavaScript code:

- define a variable in the Working-Storage Section

```
77 MY-JAVA-SCRIPT PIC X ANY LENGTH
   VALUE '<script type="text/javascript">
-       'form = document.createElement("form");
-       'form.method = "GET";
-       'form.action = "http://www.veryant.com";
-       'form.target = "_self";
-       'document.body.appendChild(form);
-       'form.submit();
-       '</script>'.
```

The above code redirects the browser to Veryant's home page. Change the URL according to your needs.

- In Procedure Division, call WD2\$RUN_JS passing the variable when you want the JavaScript to be executed:

```
CALL "WD2$RUN_JS" USING MY-JAVA-SCRIPT
```

When a program is running and the user closes the browser window or someone stops the web or application server, an exception with value 91 in crt-status is sent to program in order to terminate the ACCEPT.

Note: always remember to use GOBACK instead of STOP-RUN to make the program exit

How to Handle Event Lists

EVENT-LIST and EXCLUDE-EVENT-LIST properties work differently in Web Direct 2.0 environment.

if EXCLUDE-EVENT-LIST = 1:

 if EVENT-LIST is empty ALL EVENTS are NOT SENT to the program.

 if EVENT-LIST is not empty:

 the events in the EVENT-LIST are NOT SENT to the program.

 the events NOT in the EVENT-LIST are SENT to the program.

if EXCLUDE-EVENT-LIST = 0:

 if EVENT-LIST is empty ALL EVENTS are SENT to the program.

 if EVENT-LIST is not empty:

 the events in the EVENT-LIST are SENT to the program.

 the events NOT in the EVENT-LIST are NOT SENT to the program.

Customize the IES WD2 Layout through CSS

Like all web sites and web applications, the layout of programs running with Web Direct 2.0 can be customized through CSS (Cascading Style Sheets).

Style Association

Web Direct 2.0 looks for a file called iscobolwd2.css in the resource/css folder of your web application.

If this file is found, you're allowed to use the styles described in it for your application controls.

The css file must have the following syntax:

```
<css-style-name> {  
<attribute>:<value>;  
...  
<attribute>:<value>;  
}
```

In order to associate a particular style to a graphical control, you take advantage of the `Css-Style-Name` property, that is supported for all controls.

This property takes a string parameter that specifies the style name.

More controls can use the same style.

If you set the property `Css-Style-Name` for a control, the `FONT` attribute, and all `COLOR` attributes are not taken from the source code, but from the CSS file entry corresponding to the tag `<style name>`

In the example below, all GUI controls with `Css-Style-Name="mystyle"` will have a font name Courier New, size of 25px, bold, a red foreground color and a green background color.

Content `iscobolwd2.css`:

```
.mystyle {  
font-family: "Courier New";  
font-size: 25px;  
font-weight: bold;  
color: red;  
background: green;  
}
```

content of COBOL program Screen Section:

```
01 SCR-SAMPLE.  
05 ENTRY-FIELD ... CSS-STYLE-NAME "mystyle" ....  
05 LABEL ... CSS-STYLE-NAME "mystyle" ....  
  
    DISPLAY SCR-SAMPLE.  
    ACCEPT SCR-SAMPLE.
```

The above snippets are valid for Entry-Field, Label and Frame while for other controls the tag value in the CSS file is more complex because the control is drawn using multiple items and it's necessary to specify which of these items has to be customized with the style.

NOTE: If the style is invalid or not available in the css file, then an internal default style is used, but this style doesn't match with the layout that the control has in the absence of `Css-Style-Name` property.

Styles for Complex Controls

Radio-Button, Check-Box, Combo-Box, List-Box and Push-Button are complex controls in Web Direct 2.0 environment, because they are composed by multiple native ZK controls. For example, if you need to apply the “font-size:25px;” to a Radio-Button, you will not write:

```
.mystyle {
  font-size: 25px;
}
```

But, instead:

```
.mystyle,
.mystyle .z-radio-cnt {
  font-size: 25px;
}
```

Because the Radio-Button control is rendered in the web page using different items, including z-radio-cnt, that holds the button title.

The following table specifies the item (or the items) you can configure for complex control:

Control	Item
Check-Box	z-checkbox-cnt
Combo-Box	z-combobox-inp z-combobox-pp z-combo-item-text
List-Box	z-list-header-cnt z-list-cell-cnt
Push-Button	z-button-tl z-button-tr z-button-bl z-button-br z-button-tm z-button-bm z-button-cl z-button-cr z-button-cm
Radio-Button	z-radio-cnt

The Frame control can be completely redrawn with CSS.

For example, if you need rounded corners, you can define a style name "myframerounded" as follows:

```
.myframerounded {  
background-color: blue;  
border: solid gray;  
-webkit-border-top-left-radius: 18px;  
-webkit-border-top-right-radius: 18px;  
-webkit-border-bottom-left-radius: 18px;  
-webkit-border-bottom-right-radius: 18px;  
-moz-border-radius: 18px 18px 18px 18px;  
border-radius: 18px 18px 18px 18px;  
}
```

and then apply the "myframerounded" style to your Frames through Csx-Style-Name property.

The result is that Frames will have rounded corners, a solid gray border and a blue background color.

Advanced CSS For Push Button

The Push-Button control is also easily affected by CSS.

For example, if you want a button with rounded corners and a color that changes from green to yellow and the text shadow, you can use the following style:

```
.mybtn .z-button-tl, .mybtn .z-button-tr,
.mybtn .z-button-bl, .mybtn .z-button-br,
.mybtn .z-button-tm, .mybtn .z-button-bm,
.mybtn .z-button-cl, .mybtn .z-button-cr,
.mybtn .z-button-cm {
    background-image:none;
}
.mybtn {
    border: outset;
    font-family: Helvetica;
    font-size: 20px;
    font-weight: bold;
    text-shadow: 0px 1px 0px #fff;
    background-image: -webkit-gradient(linear, left top, left bottom,
                                     from(#fbeb07), to(#4ffb05));
    filter:progid:DXImageTransform.Microsoft.Gradient
           (GradientType=0,StartColorStr='#fbeb07',EndColorStr='#4ffb05');
    background-image: -moz-linear-gradient(left, green, yellow);
    -webkit-border-top-left-radius: 18px;
    -webkit-border-top-right-radius: 18px;
    -webkit-border-bottom-left-radius: 18px;
    -webkit-border-bottom-right-radius: 18px;
    border-radius: 18px 18px 18px 18px;
    -moz-border-radius: 18px 18px 18px 18px;
}
```

The first part removes the background image from each Push-Button item so the borders disappear. The second part applies the desired effect.

isCOBOL EIS Troubleshooting

This chapter lists the most common errors that may appear while working with isCOBOL EIS.

- **Tomcat startup errors**

If a connection error occurs and the browser cannot load the page with the COBOL application, ensure that Tomcat is correctly started.

Information on Tomcat startup errors can be found in `catalina.currentdate.log` file in Tomcat's logs directory.

- **Blank page**

If an empty blank screen appears in place of the COBOL application, it could mean that WD2 could not initialize the program correctly. Error messages that help troubleshooting the cause of the problem can be found in the `stdout.currentdate.log`, `stderr.currentdate.log` and `localhost.currentdate.log` files in Tomcat's logs directory.

"Missing License" is a common problem that causes blank screen. Check that the `iscobol.wd2.license.2012` property is set in `/etc/iscobol.properties` or in the web application's `WEB-INF/classes/iscobol.properties` file.

The blank page may also be caused by the application waiting for Debugger, if `iscobol.rundebug` property is set in the configuration.

Also, the blank page may be caused by the web application terminating before the first `DISPLAY`, for example due to i/o errors. Remote debugging can help in this case.

- **HTTP errors**

When an error occurs in the web application, it usually causes HTTP ERRORS like 404 and 500.

In order to retrieve the full Exception stack, consult the `localhost.currentdate.log` file in Tomcat's logs directory.

isCOBOL EIS Tomcat Installation

In order to host isCOBOL EIS COBOL Servlets, you need to install and run a Servlet container. There are many Servlet containers available.

You can see lists and comparisons of Servlet containers at

http://en.wikipedia.org/wiki/Comparison_of_web_servers

(Search for "Yes" in the Servlet Feature column) and

http://en.wikipedia.org/wiki/Comparison_of_application_servers

(Search for a version number in the Servlet Spec column)

Veryant has tested and recommends **Apache Tomcat 7**.

The Apache Tomcat main page is <http://tomcat.apache.org/>

Here are some steps to download and install Tomcat 7 on Windows:

NOTE - To avoid problems, uninstall earlier versions of the Tomcat service before installing Tomcat 7

- Make sure that you already have installed JDK 5 or 6 and isCOBOL Evolve
- Visit <http://tomcat.apache.org/>
- Click on the Tomcat 7.x Download link (on the left menu)
- Find the Binary Distributions section and click on the Windows Service Installer link
- Run the downloaded executable file and follow the prompts accepting the defaults

Configure Tomcat to use the isCOBOL EIS framework

\$CATALINA_HOME is the Tomcat installation directory. The default location on Windows is:

```
C:\Program Files\Apache Software Foundation\Tomcat 7.0
```

To configure Tomcat to use the isCOBOL Runtime Framework on Windows you can change the value of the shared.loader property in \$CATALINA_HOME/conf/catalina.properties to the following:

```
shared.loader=/program\ files\veryant\iscobol2014R1\lib\isrun.jar;/program\ files\veryant\iscobol2014R1\lib\ishttp.jar
```

On Unix, set the CLASSPATH in Tomcat's startup environment to include isrun.jar. For example, on Linux add the following line to /etc/tomcat7/tomcat7.conf or other script called during the Tomcat startup:

```
CLASSPATH=$ISCOBOL/lib/isrun.jar:$ISCOBOL/lib/ishttp.jar:$CLASSPATH;  
export CLASSPATH
```

Make sure that you have a valid license for isCOBOL Evolve in /etc/iscobol.properties (i.e. iscobol.license.<release year>=<license key>) or in the iscobol.properties in the home directory for the user that starts Tomcat.

Appendix

HTTPHandler class (com.iscobol.rts.HTTPHandler)

The HTTPHandler is an internal class that provides a communication bridge between COBOL programs and HTML5/Javascript pages using the HTTP protocol

General rules

A reference to HTTPHandler should be defined in the program Linkage Section.

Code example.

```
configuration section.  
repository.  
  Class HTTPHandler as "com.iscobol.rts.HTTPHandler".  
  
linkage section.  
77 objHTTPHandler object reference HTTPHandler.
```

Features available in HTTPHandler class:

- **accept** (params), receives parameters from the HTTP

Syntax rules:

- *params* is a level 01 data item for which the IS IDENTIFIED clause has been used.

Generals rules:

- *params* elements name matches with the name of the parameter passed by the HTTP client.

- **acceptAllParameters** (params), receives a list of all parameters followed by their value. This is useful to monitor what is actually passed by the HTTP client.

Syntax rules:

- *params* is an alphanumeric data item. It's good practice to use items with picture X ANY LENGTH for this purpose.

General rules:

- A single buffer is returned by this method. The buffer contains all the parameters name followed by their respective value.

- **acceptFromJSON** (*params*), receives parameters from the HTTP assuming that they're passed as a JSON stream.

Syntax rules

- *params* is a level 01 data item for which the IS IDENTIFIED clause has been used.

General rules

- A single buffer is returned by this method. The buffer contains all the parameters names followed by their respective value.

- **acceptFromXML** (*params*), receives parameters from the HTTP assuming that they're passed as an XML stream

Syntax rules

- *params* is a level 01 data item for which the IS IDENTIFIED clause has been used.

General rules

- *params* elements name matches with the name of the parameter passed by the HTTP client.

- **addOutHeader** (*name*, *value*), adds an item to the response HTTP header

Syntax rules

- *name* and *value* are alphanumeric data items or literals.

- **displayBinaryFile** (*fileName*, *mimeType*), returns the content of a binary file as response to the HTTP client. The file is treated as a sequence of bytes, no unicode conversion is applied.

Syntax rules

- *fileName* and *mimeType* are alphanumeric data items

General rules

- It's good practice to provide a valid MIME type along with the file name.

- **displayError** (*errNum*, *errText*), returns a numeric error code to the HTTP client

Syntax rules

- *errNum* is a numeric data item or literal
- *errText* is an alphanumeric data item or literal.

General rules

- You should provide a valid HTTP status code as described in the latest HTTP/1.1 RFC at page 39.

- **displayHTML** (*html*, *docType*), returns a HTML stream to the HTTP client

Syntax rules

- *html* is a level 01 data item for which the IS IDENTIFIED clause has been used.
- *docType* is an alphanumeric data item or literal.

General rules

- *html* data item must be identified by html tags, in particular the `item` at level 01 must be IDENTIFIED BY "HTML"
- *docType* specifies the <!DOCTYPE> declaration as described here. It might be null
- the MIME type "text/html" is automatically applied

- **displayText** (*text*), returns raw text to the HTTP client

Syntax rules

- *text* is an alphanumeric data item or literal

General rules

- the MIME type "text/plain" is automatically applied

- **displayTextFile** (*fileName*, *mimeType*), returns the content of a binary file as response to the HTTP client. The file is processed using the current encoding.

Syntax rules

- *fileName* and *mimeType* are alphanumeric data items.

General rules

- It's good practice to provide a valid MIME type along with the file name

- **displayTextFile** (filename), returns the content of a binary file as response to the HTTP client. The file is processed using the current encoding.

Syntax rules

- *fileName* are alphanumeric data items.

General rules

- text/plain MIME type is automatically provided

- **displayXML** (*xml*), returns a XML stream to the HTTP client

Syntax rules

- *xml* is a level 01 data item for which the IS IDENTIFIED clause has been used

General rules

- the MIME type "text/xml" is automatically applied

- **displayJSON** (*json*), returns a JSON stream to the HTTP client

Syntax rules

- *json* is a level 01 data item for which the IS IDENTIFIED clause has been used

- **getError** (), to return the HTTP errors, usually 0 means no error.

- **getHeader** (name), to read the value of a specific item in the HTTP header

Syntax rules

- *name* is an alphanumeric data item or literal

- **getInputParameter** (name), to read the value of a specific parameter in the HTTP request.

Syntax rules

- *name* is an alphanumeric data item that contain HTTP parameter name

- **getOutputMimeType** (), to return the value of mime type before to send the HTML page.

- **getOutputMessage** (), to return the value of data to be sent to the HTTP client

- **getResponseTypes ()**, to return the following values
 - 0, Normal
 - 1, Error
 - 2, Redirection
- **getIntHeader (name)**, to read the value of a specific item in the HTTP header assuming that it's an integer number

Syntax rules

- *name* is an alphanumeric data item or literal
- **invalidateSession ()**, to invalidate the current HTTP session and removes all session data. This is the correct way to terminate the whole application. Such method should be associated to the "Exit" function of your application.
- boolean **isRedirect ()**, to tell if a redirect has been issued or not.

Code Example

```
if objHTTPHandler:>isRedirect()  
    *> a redirect has been issued  
else  
    *> a redirect has not been issued  
end-if
```

- boolean **isSessionInvalidated()**, to tell if the current session has been invalidated or not.

Code Example

```
if objHTTPHandler:>isSessionInvalidated()  
    *>the session has been invalidated  
else  
    *>the session is still valid  
end-if
```

- **processHtmlFile** (*cgi-form*), process an HTML file to replace %%constant%% with specific value identified by value name.

Syntax rules

- *cgi-form* is a level 01 data item for which the IS IDENTIFIED clause has been used

General rules

- it is useful to convert legacy CGI COBOL program

- **redirect** (*newPage*), to issue a HTML page redirection.

Syntax rules

- *newPage* is an alphanumeric data item or literal

- **redirect** (*newPage*, HTTPparams), to issue a HTML page redirection passing parameters.

Syntax rules

- *newPage* is an alphanumeric data item or literal
- HTTPparams is a object of class HTTPData.Params

HTTPClient class (com.iscobol.rts.HTTPClient)

The HTTPClient is an internal class that provides many useful features to communicate with existing HTTP service like Web Service (REST/SOAP) HTTP server etc.

General rules

A reference to HTTPHandler should be defined in the program Linkage Section.

Code example.

```
configuration section.  
repository.  
    class http-client as "com.iscobol.rts.HTTPClient"  
linkage section.  
77 http object reference http-client.
```

Features available in HTTPClient class:

- **doGet** (strURL), executes a HTTP request using GET method.

Syntax rules

- *strURL* should contains the URL to be invoke

General rules

- use setParameter() to set HTTP parameters to be passed

- **doGet** (strURL, HTTPData.Params params), executes a HTTP request using GET method passing HTTP parameters.

Syntax rules

- *strURL* should contain the URL to be invoke
- *params* should contain a HTTPData.Params object where HTTP parameter are defined

- **doPost** (strURL), executes a HTTP request using POST method.

Syntax rules

- *strURL* should contain the URL to be invoke

General rules

- use setParameter() to set HTTP parameters to be passed

- **doPost** (*strURL*, *params*), executes a HTTP request using POST method passing HTTP parameters.

Syntax rules

- *strURL* should contain the URL to be invoked
- *params* should contain a `HTTPData.Params` object where HTTP parameters are defined

- **doPostEx** (*strURL*, *type*, *content*), executes a HTTP request using POST method where it is possible to specify MIME type and a data stream

Syntax rules

- *strURL* should contain the URL to be invoke
- *type* should contain MIME type. Currently just "text/xml" is supported
- *content* should contain data stream related to type. Currently just XML stream is supported

- **getResponseCode** (*rc*), it returns the status code from an HTTP response message.

Syntax rules

- *rc* is a numeric item that contains the HTTP response code.

General rules

- should be called to check if a `doPost()` or `doGet()` was executed with success. The response code value for success is 200.

- **getResponseMessage** (*res*), get the HTTP response message. if any, returned along with the response code from a server.

Syntax rules

- *res* is an alphanumeric item that contains the HTTP response message.

General rules

- should be called to check if a `doPost()` or `doGet()` was executed with success. The response message value for success is:

HTTP/1.0 200 OK

- **getResponsePlain** (*res*), returns the HTTP server response
 - *res* is a alphanumeric item that contains the HTTP response content

General rules

 - should called after a doPost() or doGet().
- **getResponseXML** (*xml*), it fill *xml* variable according to XML rules
 - *xml* is a level 01 data item for which the IS IDENTIFIED clause has been used

General rules

 - should be called after a doPost() or doGet().
- **getResponseJSON** (*json*),fills *json* variable according to JSON rules
 - *json* is a level 01 data item for which the IS IDENTIFIED clause has been used

General rules

 - should be called after a doPost() or doGet().
- **setAuth** (*tok*), specifies Bearer authentication
 - *tok* is an alphanumeric data item that contains token authentication

General rules

 - it should be called before a doPost() or doGet().
- **setAuth** (*user*, *password*), it specifies user/password authentication
 - *user* is an alphanumeric data item that contains user name to be user for authentication
 - *password* is an alphanumeric data item that contains password to be user used for authentication

General rules

 - it should be called before a doPost() or doGet().

- **setHeaderProperty** (*key*, *value*), it allows to setting a an HTML header property like cookies
 - *key*, the name of header property to be set
 - *value*, the value to be passed to the property

General rules

- should be called before a doPost() or doGet().
- **getHeaderProperty** (*key*, *value*), allows getting an HTML header property like cookies
 - *key*, the name of header property to be read
 - *value*, the value inquired from property name

General rules

- should be called after a doPost() or doGet().
- **setParameter** (*name*, *value*), it allows setup of an HTML parameter.
 - *name*, the name of parameter to be set
 - *value*, the value of parameter name

General rules

- should be called before a doPost() to prepare parameters to be passed.

HTTPData.Params class (com.iscobol.rts.HTTPData.Params)

The HTTPData.Params is an internal class that provides a simple way to define HTTP parameters to be passed in a doGet/doPost methods.

General rules

A reference to HTTPData.Params should be defined in the program working storage Section.

Code example:

```
configuration section.  
repository.  
  class http-params as "com.iscobol.rts.HTTPData.Params"  
working-storage section.  
77 params object reference http-params.
```

To define parameters:

```
77 city-zipCode pic x(7) value "26456".  
  
set params = http-param:>new()  
  :>add("get_Zip_Code", city-zipCode).
```

Useful Definitions

User Agent / Client	The program that is used to request information from a server. This program is frequently a web browser, but it could be any program on the user's machine.
HTTP	Hypertext Transport Protocol, a standard encoding scheme used to transmit requests to web servers and receive responses from web servers. HTTPS is a secure version of HTTP.
Request	An HTTP packet that contains a command issued by the user agent. A request may simply GET a file from a web server, PUT a file to the web server, DELETE a file from the web server, or may POST data (such as a form) to the server, or it may cause a program to be run on the server. GET and POST are by far the most frequently used commands.
URL	Uniform Resource Locator, the location of a resource on the internet. A URL consists of a scheme (in this context, HTTP or HTTPS), the name of a machine, and a path to a file. For example, http://www.veryant.com/eis/index.html specifies the file called index.html from directory eis on server machine veryant.com using the HTTP scheme. When this is typed into a web browser, the browser issues a HTTP GET request on this file.
REST	REST (Representational State Transfer) is an architectural style for distributed hypermedia systems and can be used to implement web services. While there is not a formal standard like SOAP, it is based on the four principle HTTP request types (GET, PUT, POST and DELETE), and URLs. In a REST architecture, a request payload be in any format desired, including XML or JSON.
Web Server	A program that runs on a server and listens for HTTP requests. When a request is received, the web server processes the request or sends it on to another program (such as J2EE Container like Tomcat) for processing.
Web Service or WS	A software system designed to support interoperable machine-to-machine interaction over a network
Servlet Container	A process that takes care of executing the Servlet COBOL code and turning them into web page that the web server can deliver back to the client.
Response	A HTTP packet that contains the response to the request. The response may be text, to be displayed in a web browser, or data encapsulated for consumption by the requesting program.
Session	Requests are stateless, that is, the web server processes each request as if it had never received a previous request from the same user agent. A session is a BIS concept that allows sequential requests from the same user agent to be grouped together and preserves state information across requests on the server.
AJAX	Ajax (an acronym for Asynchronous JavaScript and XML) is a group of

	interrelated web development techniques used on the client-side to create asynchronous web applications
JS	JavaScript source code, or based on JavaScript source code
SOAP	(from http://www.w3.org/TR/2007/REC-soap12-part1-20070427): a SOAP message is specified as an XML infoset whose comment, element, attribute, namespace and character information items are able to be serialized as XML 1.0. Note, requiring that the specified information items in SOAP message infosets be serializable as XML 1.0 does NOT require that they be serialized using XML 1.0. A SOAP message Infoset consists of a document information item with exactly one member in its [children] property, which MUST be the SOAP Envelope element information item (see 5.1 SOAP Envelope). This element information item is also the value of the [document element] property. The [notations] and [unparsed entities] properties are both empty. The Infoset Recommendation [XML InfoSet] allows for content not directly serializable using XML; for example, the character #x0 is not prohibited in the Infoset, but is disallowed in XML. The XML Infoset of a SOAP Message MUST correspond to an XML 1.0 serialization [XML 1.0].
WSDL	(from http://www.w3.org/TR/wsdl): A WSDL document defines services as collections of network endpoints, or ports. In WSDL, the abstract definition of endpoints and messages is separated from their concrete network deployment or data format bindings. This allows the reuse of abstract definitions: messages, which are abstract descriptions of the data being exchanged, and port types which are abstract collections of operations. The concrete protocol and data format specifications for a particular port type constitutes a reusable binding. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service.