# Building a REST service with integrated web services server for IBM i: Part 3

## Deploy a RESTful application using multiple HTTP methods

By Nadir Amra, IBM Software Engineer
Published 08 May 2015 (updated 05 May 2024)

**Abstract:** Rapidly changing application environments require a flexible mechanism to exchange data between different application tiers. Representational State Transfer (REST) has gained widespread acceptance across the Web as the interface of choice for mobile and interactive applications.

You may already be using integrated web services server to expose ILE programs and service programs as SOAP-based web services. This series of articles introduces a powerful new feature of the integrated web services server – the ability to deploy ILE programs and services programs as RESTful web services. In this third installment, you will learn how to deploy an application as a RESTful web service using multiple HTTP methods.

# Introduction

For several years now IBM i users have had the ability to deploy ILE programs and services programs as web services based on the SOAP protocol using the integrated web services server support that is part of the operating system. REST web services were not supported by the integrated web services server, until now.

This article is the third in a series of articles about the integrated web services server REST support.

- Part one starts out by explaining the basic concepts behind REST web services and how the integrated web services server supports REST services.
- Part two takes you through the steps of deploying a simple ILE application as a RESTful web service.
- In part three, we take you through the steps of deploying a more complex ILE application that uses more of the REST features.

# Prerequisites

## *Software*

To get all the PTFs required by the integrated web services server in support of REST, you will need to load the latest HTTP Group PTF. The IBM Support web page *IBM i Group PTFs with level* lists the HTTP group PTFs for each of the supported releases of the IBM i operating system.

**Note**: The steps in this article were performed on IBM i 7.3. Panels may look different if you are on an older or newer release. And if you are on an older release, some features discussed in this article may be unsupported on that release.

## *Assumptions*

Before reading this article, you should have read part one of the article in the series in order to have a basic understanding of REST principles and the terminology used.

# The RESTful application

The example we will use in this discussion is a sample Student Registration Application (SRA). The student registration management functions provided in this sample SRA application enable you to:

- Register new students
- Edit registered student information
- List registered students
- Remove student registrations

SRA consists of a service program, STUDENTRSC, that contain the RESTful services (i.e. procedures) that provides the basic Create-Read-Update-Delete (CRUD) database logic, and a database file, STUDENTDB, where student records are stored.

Listing 1 shows the prototypes of the procedures in the SRA application that are exported from the STUDENTRSC service program (i.e. resource). The full source of the application can be found at the end of this article in Code Listings.

## Listing 1. Partial listing of SRA application

```
ctl-opt nomain pgminfo(*pcml:*module:*dclcase);

dcl-ds studentRec qualified template;
  studentID       char(9);
  firstName       varchar(50);
  lastName        varchar(50);
  gender          char(10);
end-ds;

dcl-proc getAll export;
    dcl-pi *n;
      students_LENGTH   int(10);
      students          likeds(studentRec) dim(1000) options(*varsize);
      httpStatus        int(10);
      httpHeaders       char(100) dim(10);
    end-pi;
.
.
.
End-proc;

dcl-proc getByID export;
    dcl-pi *n;
      studentID         char(9);
      student           likeds(studentRec);
      httpStatus        int(10);
      httpHeaders       char(100) dim(10);
    end-pi;
.
.
.
end-proc;

dcl-proc create export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
      httpHeaders           char(100) dim(10);
    end-pi;
.
.
.
end-proc;

dcl-proc update export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
```

```
    end-pi;
.
.
.
end-proc;

dcl-proc remove export;
    dcl-pi *n;
      studentID             char(9);
      httpStatus            int(10);
    end-pi;
.
.
.
end-proc;
```

The resource program object defines five procedures:

- getAll: get all students.
- getById: get a particular student using their student ID.
- create: create a new student record.
- update: update a student record.
- remove: delete a particular student record.

You should also note that a data structure template (`studentRec`) is defined which defines the fields in the database file. You can optionally define the data structure via an externally described data structure, but the decision to not use an externally defined data structure is due to wanting to control the case of the XML element names and/or JSON field names, which are obtained from the generated program interface information. Previously names were all capitalized. However, a new feature was introduced and PTF'ed to IBM i 7.1 (PTF SI55340) and i 7.2 (PTF SI55442) that allows you to control the case by specifying the *DCLCASE parameter for the PGMINFO Control specification keyword. When specified, the names in the program interface information will be generated in the same case as the names defined in the RPG source file.

## Things to get done before deployment

As we have done in Part 2 of the series, we need figure out things before deploying the RESTful web service. To summarize, when deploying a RESTful web service, you should have answers to the following questions at the bare minimum:

1. How do I want the URIs to look like?
2. What HTTP methods will the resource support?
3. What incoming content types should be supported?
4. What type of data should be returned?

In the following sections go through these basic questions in the context of the application that will be deployed. Table 2 shows a summary of the mappings that we want between HTTP methods and URIs for the SRA application.

### Table 2. HTTP method and URI mappings

| HTTP Method | URI | Description |
|---|---|---|
| GET | /*context-root*/students | Return all student registrations |
| GET | /*context-root*/students/{*id*} | Return student registration |
| POST | /*context-root*/students | Register a new student |
| PUT | /*context-root*/students | Update registered student |
| DELETE | /*context-root*/students/{*id*} | Remove registered student |

**Note:** The default *context-root* for an integrated web services server is `/web/service`. The context root for a server may be changed.

## *Return all student registrations*

Listing 2 shows the `getAll()` procedure. This procedure will be used to return all student registrations and will be mapped to the HTTP GET method.

### Listing 2. Procedure getAll()

```
dcl-proc getAll export;
    dcl-pi *n;
      students_LENGTH    int(10);
      students           likeds(studentRec) dim(1000) options(*varsize);
      httpStatus         int(10);
      httpHeaders        char(100) dim(10);
    end-pi;

    clear httpHeaders;
    clear students;
    students_LENGTH = 0;

    openStudentDB();

    setll *loval STUDENTDB;

    read(e) studentR;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    endif;

    dow (NOT %eof);
        students_LENGTH = students_LENGTH+1;
        students(students_LENGTH).studentID =  STUDENTID;
        students(students_LENGTH).firstName =  FIRSTNAME;
        students(students_LENGTH).lastName  =  LASTNAME;
        students(students_LENGTH).gender    =  GENDER;

        read(e) studentR;
        if (%ERROR);
           httpStatus = H_SERVERERROR;
           return;
        endif;
    enddo;
```

```
    httpStatus = H_OK;
    httpHeaders(1) = 'Cache-Control: no-cache, no-store';

    closeStudentDB();
end-proc;
```

Notice that in the parameter list there is an array that will contains the student records and a corresponding length field that will be used to indicate how many student registration records there actually is in the array. If you do not specify the length field, the response will include empty elements which would degrade performance in both the client and server.

The `httpStatus` parameter is used for the HTTP status code that is returned to the client. On unexpected errors the HTTP status code `H_SERVERERROR` (500) is returned. Otherwise, `H_OK` (200) is returned.

The `httpHeaders` parameter is used to return HTTP headers. In this example we do not want the response cached and thus the `Cache-Control` HTTP header is set.

## *Return student registration*

Listing 3 shows the `getByID()` procedure. This procedure will be used to return a student registration and will be mapped to the HTTP GET method.

### Listing 3. Procedure getByID()

```
dcl-proc getByID export;
    dcl-pi *n;
      studentID            char(9);
      student                 likeds(studentRec);
      httpStatus            int(10);
      httpHeaders         char(100) dim(10);
    end-pi;

    clear httpHeaders;
    clear student;

    openStudentDB();

    chain(e) studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        student.studentID = studentID;
        student.firstName = firstName;
        student.lastName  = lastName;
        student.gender    = gender;

        httpStatus = H_OK;
    else;
        httpStatus = H_NOTFOUND;
    endif;
```

```
    httpHeaders(1) = 'Cache-Control: no-cache, no-store';

    closeStudentDB();
end-proc;
```

The parameter `studentID` is an input parameter that is used as a key to read the student registration from the database. If not found the HTTP status code `H_NOTFOUND` (404) is returned. The caching control HTTP header is also set so the response is not cached.

## *Register a new student*

Listing 4 shows the `create()` procedure. This procedure will be used to create a new student registration and will be mapped to the HTTP POST method.

### Listing 4. Procedure create()

```
dcl-proc create export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
      httpHeaders           char(100) dim(10);
    end-pi;

    openStudentDB();

    studentID = student.studentID;
    firstName = student.firstName;
    lastName  = student.lastName;
    gender    = student.gender;

    write(e) studentR;
    if NOT %ERROR;
        httpStatus = H_CREATED;
        // URL will need to change to your server and port
        httpHeaders(1) = 'Location: ' +
          'http://server:port/web/service/students/' + studentID;
    elseif %STATUS = ERR_DUPLICATE_WRITE;
        httpStatus = H_CONFLICT;
    else;
        httpStatus = H_SERVERERROR;
    endif;

    closeStudentDB();
end-proc;
```

If the student ID already exists in the database, an HTTP status code of `H_CONFLICT` (409). An HTTP status code of `H_CREATED` (201) is returned on a successful create. For HTTP 201 status code responses the `Location` HTTP is returned and is set to the URI of the new resource which was created by the request.

## *Update a registered student*

Listing 5 shows the `update()` procedure. This procedure will be used to update an existing student registration and will be mapped to the HTTP PUT method.

### Listing 5. Procedure update()

```
dcl-proc update export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
    end-pi;

    openStudentDB();

    chain(e) student.studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        studentID = student.studentID;
        firstName = student.firstName;
        lastName  = student.lastName;
        gender    = student.gender;

        update(e) studentR;
        if NOT %ERROR;
           httpStatus = H_NOCONTENT;
        else;
           httpStatus = H_NOTFOUND;
        endif;
    else;
        httpStatus = H_NOTFOUND;
    endif;

    closeStudentDB();
end-proc;
```

If the student ID is not found, an HTTP status code of `H_NOTFOUND` (404). Notice also on a successful update operation the HTTP status code of `H_NOCONTENT` (204) is returned since the procedure does not return any data.

## *Remove a registered student*

Listing 6 shows the `remove()` procedure. This procedure will be used to remove an existing student registration and will be mapped to the HTTP DELETE method.

### Listing 6. Procedure remove()

```
dcl-proc remove export;
    dcl-pi *n;
      studentID             char(9);
      httpStatus            int(10);
    end-pi;

    openStudentDB();
```

```
    chain(e) studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        delete(e) studentR;
        if NOT %ERROR;
           httpStatus = H_NOCONTENT;
        elseif NOT %FOUND;
           httpStatus = H_NOTFOUND;
        else;
           httpStatus = H_SERVERERROR;
        endif;
    else;
        httpStatus = H_NOTFOUND;
    endif;

    closeStudentDB();
end-proc;
```

If the student ID is not found, an HTTP status code of `H_NOTFOUND` (404). On a successful delete operation the HTTP status code of `H_NOCONTENT` (204) is returned since the procedure does not return any data.

## What you need

The example REST API developed this article assumes a database of student registrations and focuses on allowing you to retrieve, add, delete, and update these student registrations using normal REST conventions.

If you want to recreate the steps on your server, the source of the SRA application is available at the end of this article in Code Listings.

## Step 1. Set up the application database file

In this example the STUDENTDB DB file will be created in library STUDENTS. To create the library, issue the following CL command:

```
CRTLIB STUDENTS
```

To create the table, issue the following SQL command:

```
CREATE TABLE STUDENTS/STUDENTDB
("studentID"  FOR COLUMN studentID CHAR (9) NOT NULL,
  "firstName"  FOR COLUMN firstName CHAR (50) NOT NULL,
  "lastName"   FOR COLUMN lastName  CHAR (50) NOT NULL,
  "gender" FOR COLUMN gender CHAR (10) NOT NULL,
  PRIMARY KEY ( studentID ))
  RCDFMT studentr
```

To populate the table with sample student registration data, issue the following SQL command:

```
INSERT INTO STUDENTS/STUDENTDB
 (studentID, firstName, lastName, gender)
 VALUES('823M934LA', 'Nadir', 'Amra', 'Male'),
       ('826M660CF', 'John', 'Doe', 'Male'),
       ('747F023ZX', 'Jane', 'Amra', 'Female')
```

You must ensure that the user profile that will be running the service has authority to the library and database file. In this example we will be using the default user profile for the server, QWSERVICE. So, issue the following CL command:

```
CHGAUT OBJ('/qsys.lib/students.lib/studentdb.file')
       USER(QWSERVICE) DTAAUT(*RWX)
```

## Step 2. Create the integrated web services server

To deploy an ILE program object as a REST service, you need to have an integrated web services server created, and it must be version 2.6 or greater. If you have one already created, you can skip this section. If you need to create one, please see part two of this series of articles to learn how to create a server.

## Step 3. Deploy the ILE application as a RESTful web service

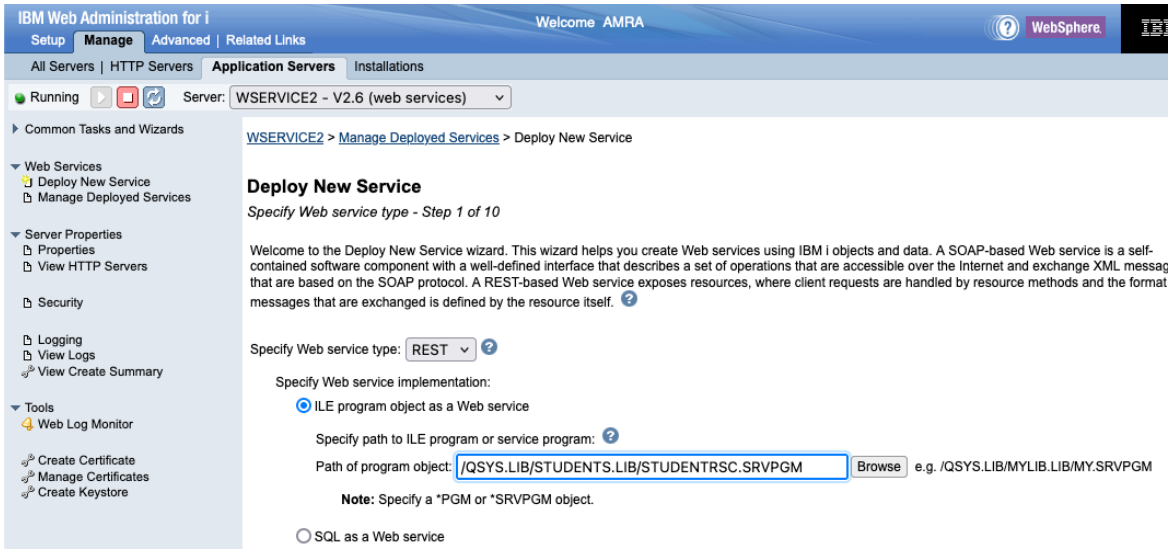
Now we deploy the SRA application service program as a RESTful web service. The service program STUDENTRSC is assumed to be in library STUDENTRSC. To create the service program, copy the source listing into a file and issue the following CL commands (the following statements assumes that the source for the SRA application is stored in "/iwsexample/studentrsc.rpgle"):

```
ADDLIBLE LIB(STUDENTS)

CRTRPGMOD MODULE(STUDENTS/STUDENTRSC)
          SRCSTMF('/iwsexample/studentrsc.rpgle')

CRTSRVPGM SRVPGM(STUDENTS/STUDENTRSC) EXPORT(*ALL)
```

### *Step 3-1. Deploy an IBM i program object as a web service*

Click on the **Deploy New Service** wizard link that is in the navigation bar. You should see the panel in Figure 1.

## Figure 1. Deploy web service – step 1



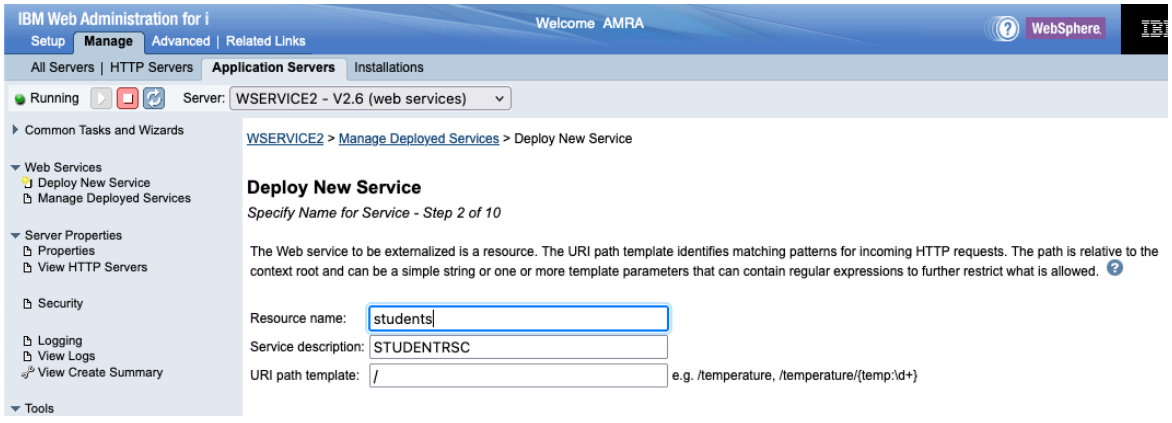

This panel gives you the option to either deploy a SOAP or REST web service. Since we are deploying a REST web service, we have selected the REST radio button. We also have inserted the path to the ILE program object that is to be deployed, which is "/QSYS.LIB/STUDENTS.LIB/STUDENTRSC.SRVPGM".

Click on the **Next** button of the form.

### *Step 3-2. Specify name for the resource (web service)*

Now we need to give the web service (i.e. resource) a meaningful service name and description. By default, the service name and description are set to the name of the selected program object (see Figure 2).

## Figure 2. Deploy web service – step 2




The resource name has been changed to `students`. In addition, you can set a URI path template for the resource. For this example, we do not need to specify anything since the path to the resource after changing the resource name is:

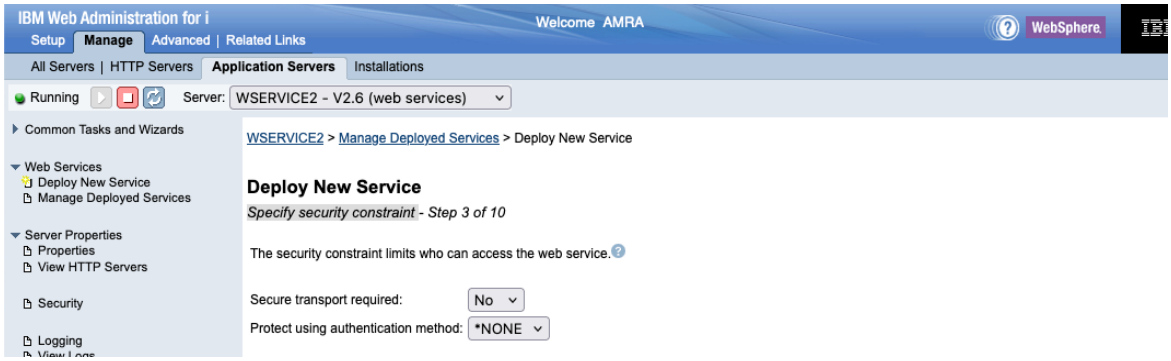
/*context-root*/students

Which is what we want. Click on the **Next** button at bottom of form.

## *Step 3-3. Specify security constraint*

The security constraint limits who can access the web service. To protect the web service, an authentication method other than *NONE needs to be specified (see Figure 3). If the web service is protected and roles have been defined, you will have the option to indicate what roles are authorized to the web service. If roles have not been defined, then all authenticated users are allowed access to the web service

### Figure 3. Deploy web service – step 3

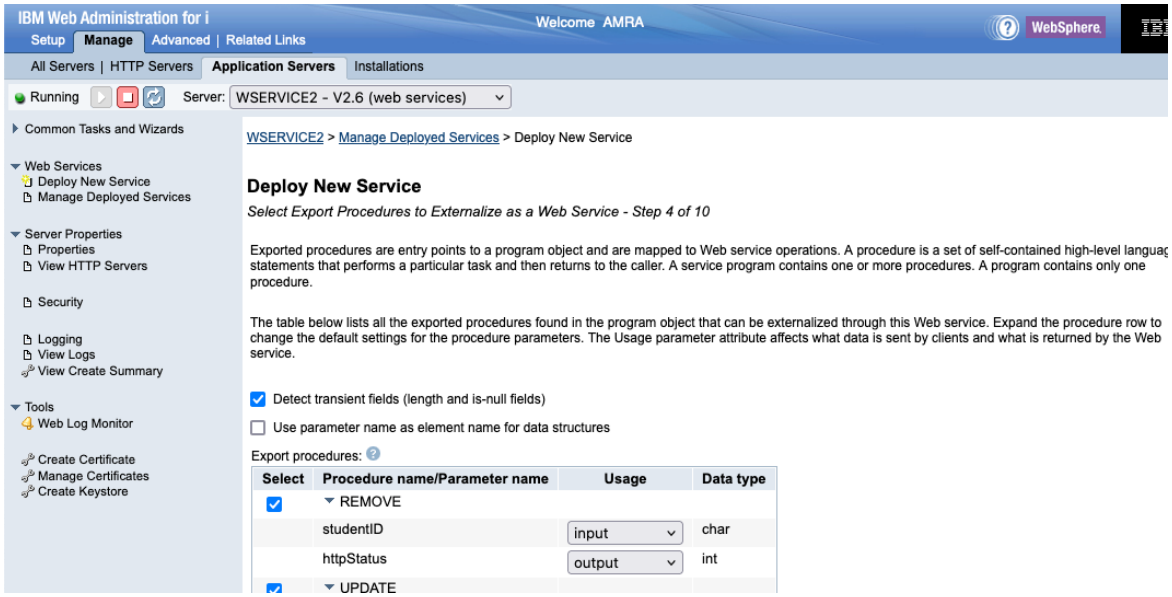



The security constraint panel is beyond the scope of this article. We accept the default values and click on the **Next** button at bottom of form.

## *Step 3-4. Select export procedures to externalize as resource methods*

The wizard will show a list of exported procedures as shown in Figure 4. For service programs (object type of *SRVPGM), there may be one or more procedures. For programs (object type of *PGM), there is only one procedure, which is the main entry point to the program. Expanding the procedure row shows the parameters for the procedure and various parameter attributes.

## Figure 4. Deploy web service – step 4



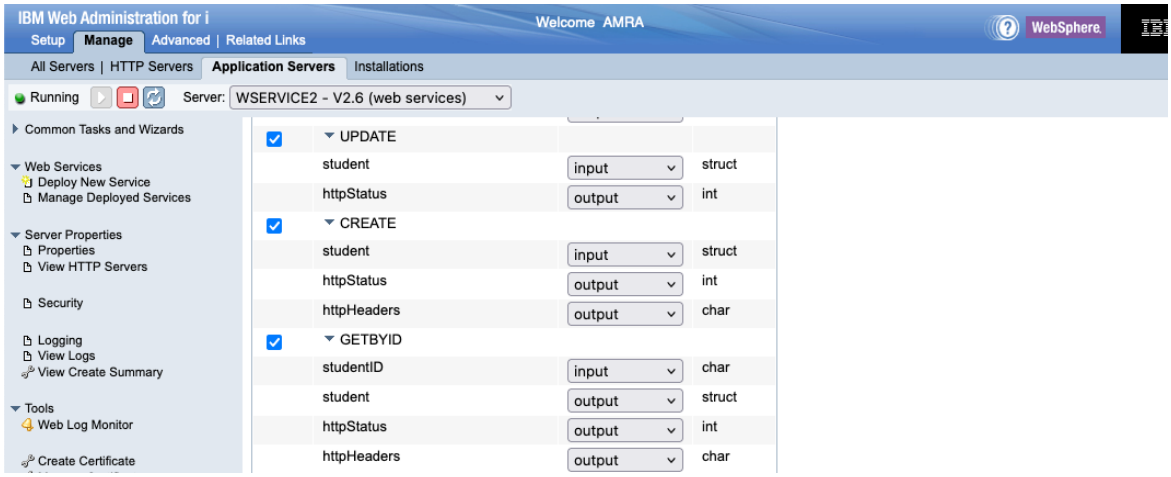

Ensure the checkbox **Detect transient fields (length and is-null fields)** is selected. When **Detect transient fields (length and is-null fields)** is selected, integrated web services support will assumed that any numeric field that immediately precedes an array field with the same name as the array field appended with _LENGTH is a length field that will be used to indicate the actual number of elements in the array. The length field is transient, that is, not considered part of the payload. Without length fields, all the elements in the array are returned. You will see an example of a length field later on in this section when we look at the getAll() procedure.

The parameter attributes are modifiable. In most cases you want to modify the parameter attributes to control what data is to be sent by web service clients and what data is to be returned in the responses to the client requests.

In Figure 4 the `REMOVE` procedure is the procedure that is to be used to remove a student registration. It has two parameters, `studentID` and `httpStatus`. The `studenID` is the identifier of the student to be removed and thus is an input parameter to the procedure. The `httpStatus` parameter is the HTTP status code to be returned in the response to the client and is designated as an output parameter.

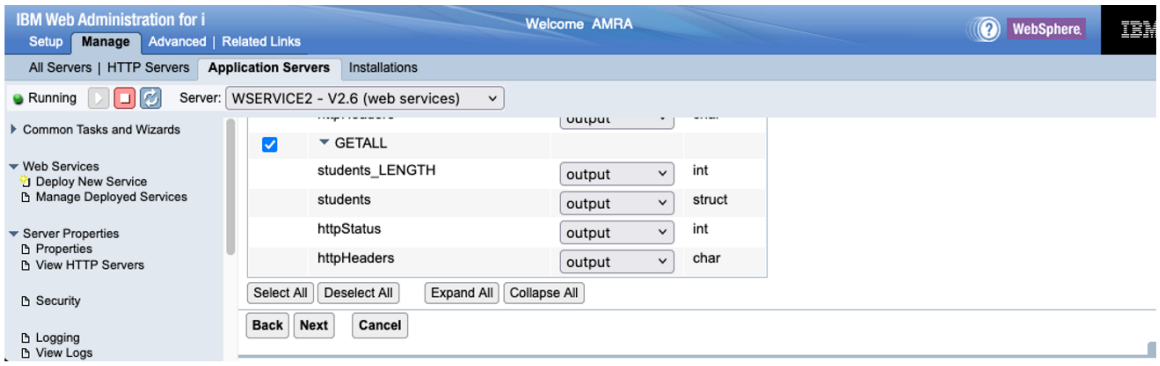
If you scroll down the exported procedure panel you see (see Figure 5) the `UPDATE`, `CREATE`, and `GETBYID` procedures. The parameters have been designated as input or output parameters.

## Figure 5. Deploy web service – step 4 (update, create, and getByID)




Scrolling down a little bit more you will find the `GETALL` procedure (see Figure 6) that returns all the student registration data in the database.

## Figure 6. Deploy web service – step 4 (getAll)



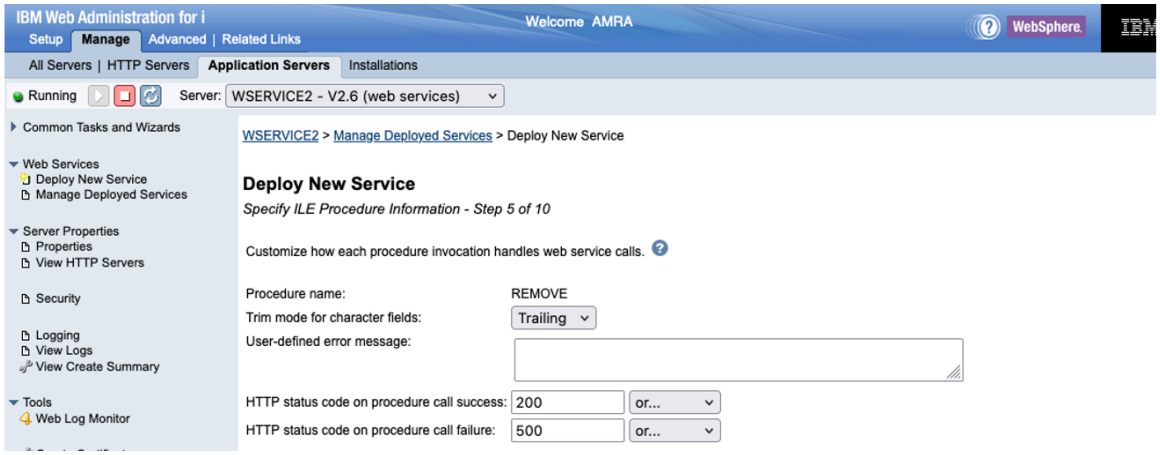

Since we want to only return the actual number of registered students, we have defined a length field parameter called `students_LENGTH`. The length field tells integrated web services support how many elements to return in the response. It will be set by the `GETALL` procedure to the actual number of registered students and thus the response will contain data for only the actual number of registered students. Without the length field, the procedure will return a response that will contains 1000 student records in which 3 student records will contain student registration data, and 995 student records will be empty.

Click on the **Next** button at bottom of form.

### *Step 3-5. Specify ILE procedure information*

This panel (Figure 7) is shown for each procedure to be deployed as a web service and allows you to indicate how each procedure invocation handles web service calls.

## Figure 7. Specify ILE procedure information – step 5 (remove)




You can indicate whether character fields are to be trimmed of blanks. In addition, you can specify a user defined messages if the call to the procedure fails and indicate what HTTP status codes to use on a procedure call success and failure.

For this example, accept the defaults for each of the procedures by pressing **Next** until you get to the **Specify resource method information** panel.

### *Step 3-6. Specify resource method information*

Before discussing this step, it is a good idea to summarize the REST information for the RESTful application that is to be deployed. Table 3 below summarizes REST information for each of the resource methods (i.e. procedures) of the SRA application.

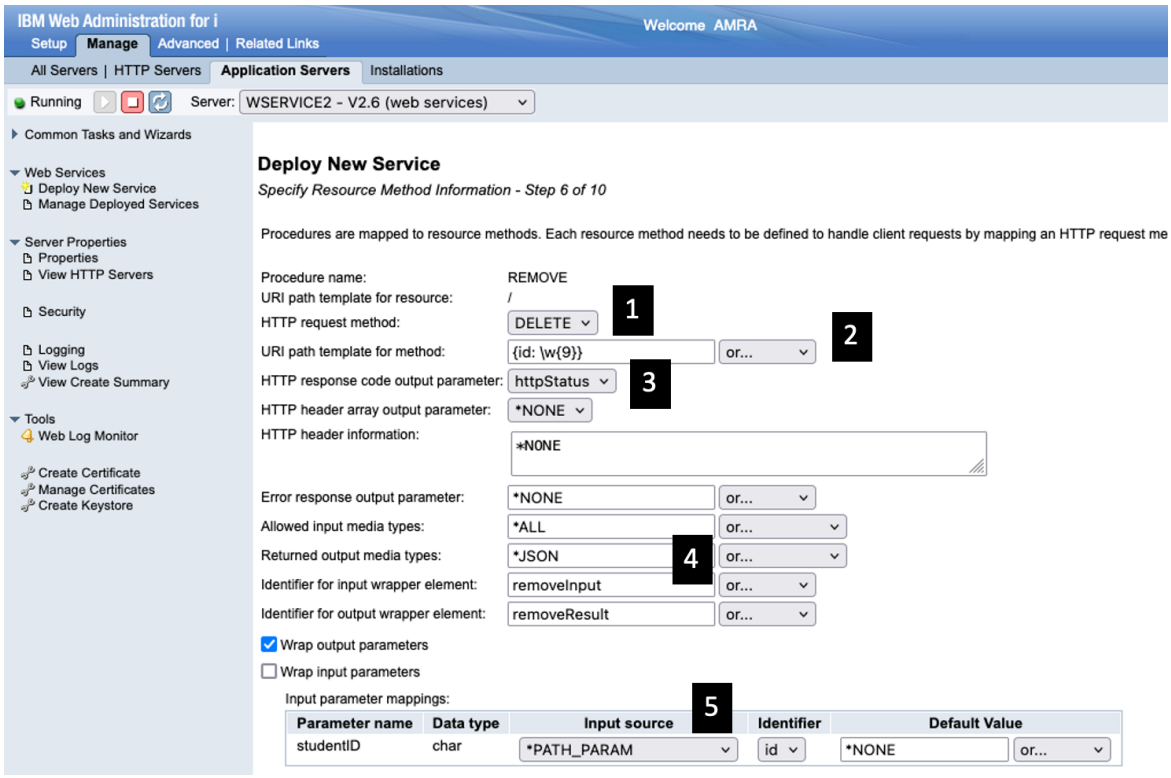
## Table 3. REST information for each procedure

| | | |
|---|---|---|
| **REMOVE** | **URL** | /*context-root*/students/*{id}* |
| | **Method** | DELETE |
| | **Request body** | None |
| | **Returns** | 204 No content |
| | | 404 Not found |
| | | 500 Server error |
| **UPDATE** | **URL** | /*context-root*/students |
| | **Method** | PUT |
| | **Request body** | JSON |
| | **Returns** | 204 No content |
| | | 404 Not found |
| | | 500 Server error |
| **CREATE** | **URL** | /*context-root*/students |
| | **Method** | POST |
| | **Request body** | JSON |
| | **Returns** | 201 Created |
| | | 409 Conflict |
| | | 500 Server error |
| **GETBYID** | **URL** | /*context-root*/students/*{id}* |
| | **Method** | GET |
| | **Request body** | None |

| | | |
|---|---|---|
| | **Returns** | 200 OK & JSON |
| | | 404 Not found |
| | | 500 Server error |
| **GETALL** | **URL** | /*context-root*/students |
| | **Method** | GET |
| | **Request body** | None |
| | **Returns** | 200 OK & JSON |
| | | 500 server error |

First procedure to be processed is the `REMOVE` procedure.

## Figure 8. Deploy web service – step 5 (REMOVE)



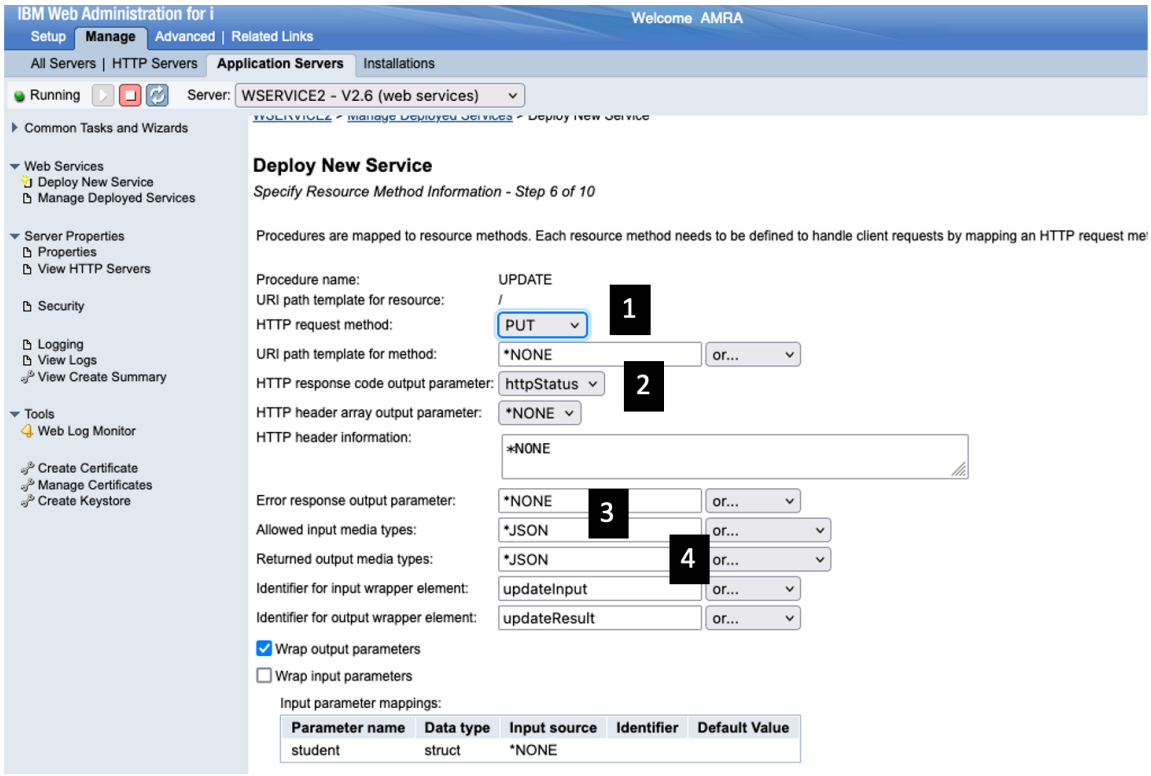

Looking at Figure 8, you will find that:

- The HTTP request method (**1**) is set to DELETE.
- Recall from Table 3 that the URI has the following format: /*context-root*/students/*{id}*. That is, the student identifier information is passed in as part of the URI. So, we specify a URI path template (**2**) so that any HTTP DELETE request that matches the URI will be passed to the `REMOVE` procedure. Notice that a regular expression `\w{9}` is used to ensure the registration ID is a word character that has a length of 9.
- We have indicated in that the `httpStatus` parameter (**3**) is to be used as the HTTP response code since the procedure will be returning a response code.

- There is no entity body returned by the procedure. However, we needed to specify something for this procedure so we chose JSON (**4**) although we could have left the default of XML or JSON.
- We want to inject the URI path variable `id` into the `studentID` parameter. We do this by specifying the input source as `*PATH_PARAM` (**5**) and select the identifier to be inserted.

Click on the **Next** button of the form to process the `UPDATE` procedure (Figure 9).

## Figure 9. Deploy web service – step 5 (UPDATE)

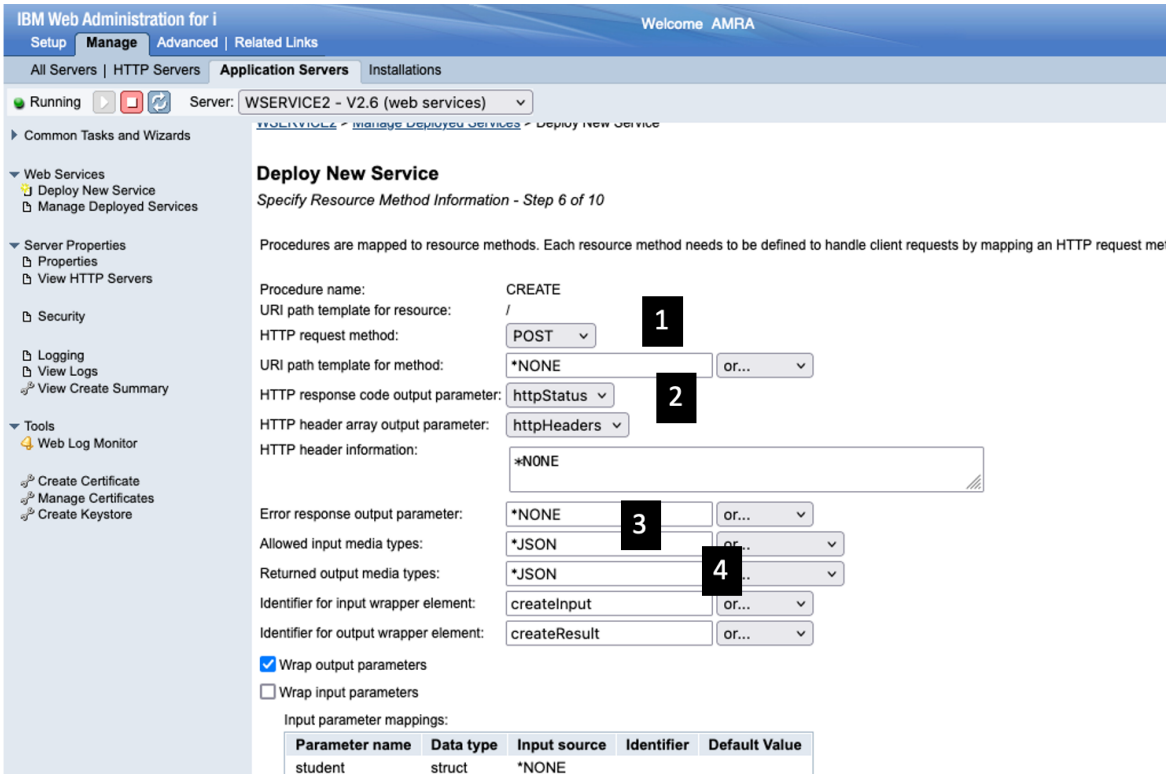



Looking at Figure 9, you will find that:

- The HTTP request method (**1**) is set to PUT.
- We have indicated in that the `httpStatus` parameter (**2**) is to be used as the HTTP response code since the procedure will be returning a response code.
- The format of the input data is JSON (**3**).
- There is no entity body returned by the procedure. However, we needed to specify something for this procedure, so we chose JSON (**4**).

Click on the **Next** button of the form to process the `CREATE` procedure (Figure 10).

## Figure 10. Deploy web service – step 5 (CREATE)

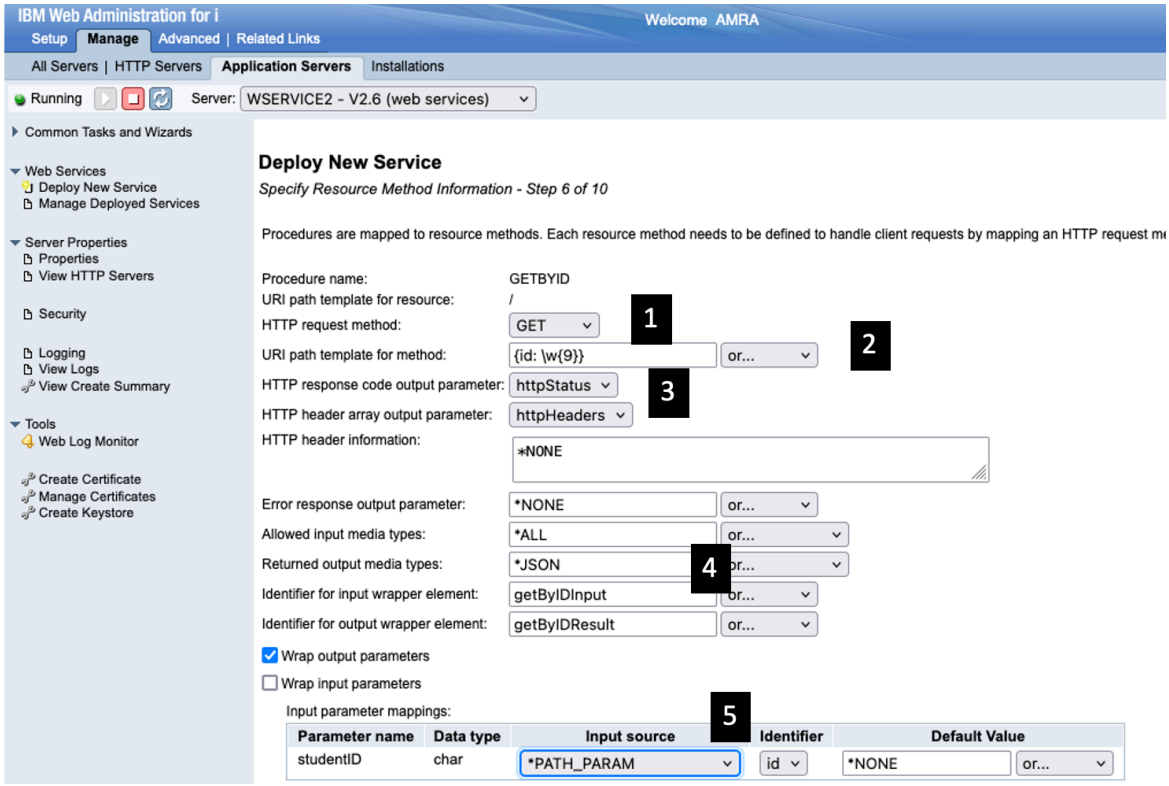



Looking at Figure 10, you will find that:

- The HTTP request method (**1**) is set to POST.
- We have indicated in that the `httpStatus` parameter (**2**) is to be used as the HTTP response code since the procedure will be returning a response code. In addition, we have indicated that the procedure will be returning HTTP headers. Recall that the HTTP Location header is set when a student registration is created successfully.
- The format of the input data is JSON (**3**).
- There is no entity body returned by the procedure. However, we needed to specify something for this procedure, so we chose JSON (**4**).

Click on the **Next** button of the form to process the `GETBYID` procedure (Figure 11).

## Figure 11. Deploy web service – step 5 (GETBYID)

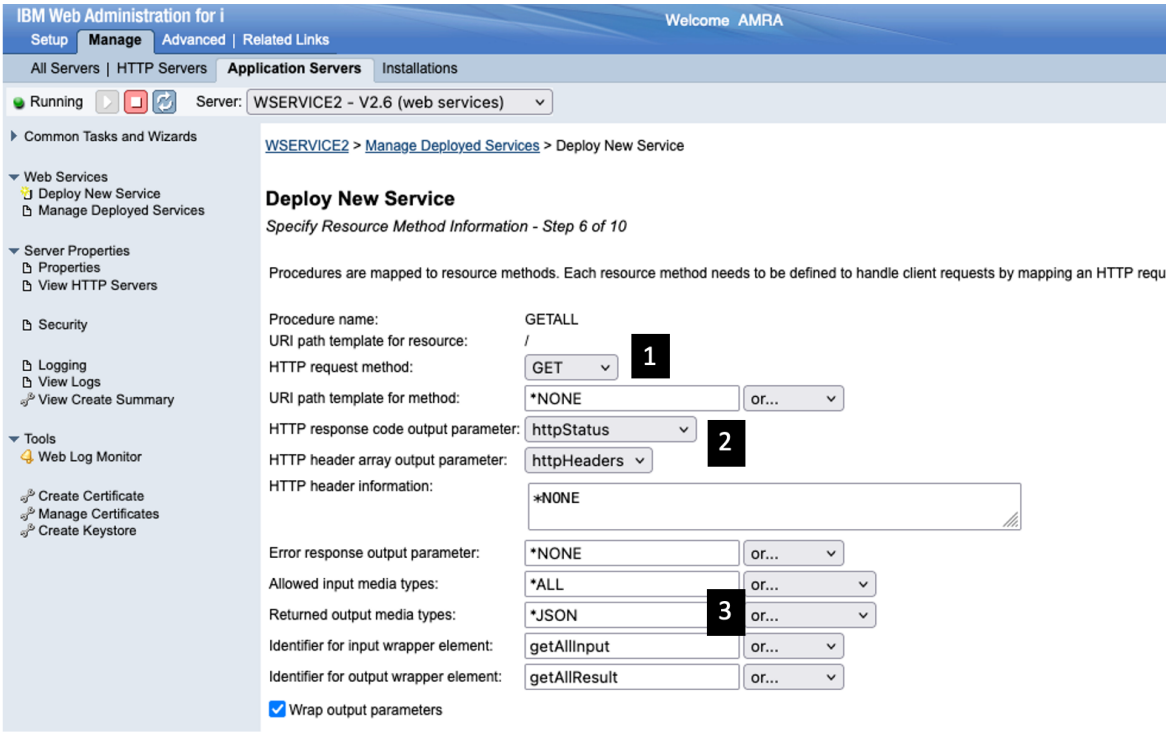



Looking at Figure 11, you will find that:

- The HTTP request method (**1**) is set to GET.
- Recall from Table 3 that the URI has the following format: /*context-root*/students/*{id}*. That is, the student identifier information is passed in as part of the URI. So, we specify a URI path template (**2**) so that any HTTP GET request that matches the URI will be passed to the `GETBYID` procedure. Notice that a regular expression `\w{9}` is used to ensure the registration ID is a word character that has a length of 9.
- We have indicated in that the `httpStatus` parameter (**3**) is to be used as the HTTP response code since the procedure will be returning a response code. In addition, we have indicated that the procedure will be returning HTTP headers. Recall that the HTTP caching header is set when a student registration is returned.
- The format of the output data is JSON (**4**).
- We want to inject the URI path variable `id` into the `studentID` parameter. We do this by specifying the input source as `*PATH_PARAM` (**5**) and select the identifier to be inserted.

Click on the **Next** button of the form to process the `GETALL` procedure (Figure 12).

## Figure 12. Deploy web service – step 5 (GETALL)




Looking at Figure 12, you will find that:

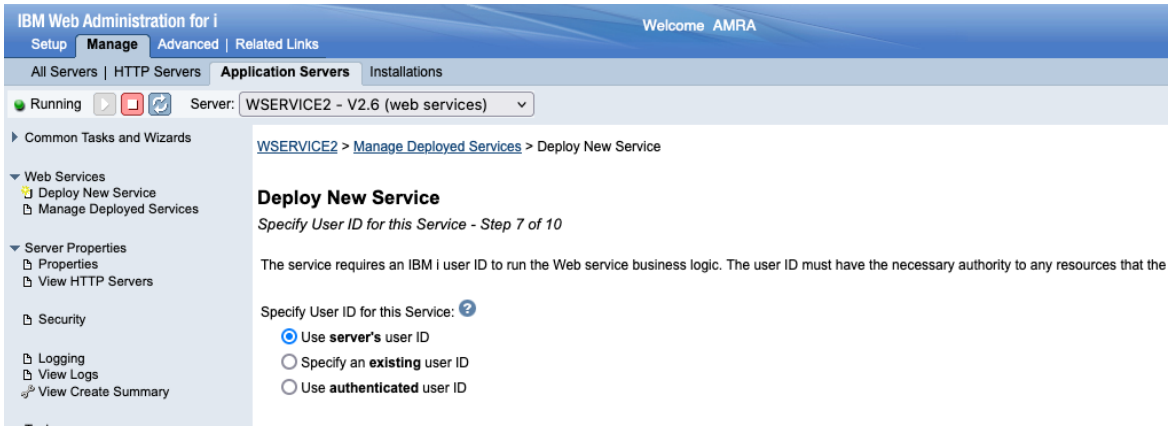
- The HTTP request method (**1**) is set to GET.
- We have indicated in that the `httpStatus` parameter (**2**) is to be used as the HTTP response code since the procedure will be returning a response code. In addition, we have indicated that the procedure will be returning HTTP headers. Recall that the HTTP caching header is set when a student registration is returned.
- The format of the output data is JSON (**3**).

At this point we are done with setting REST information. Click on the **Next** button of the form.

### *Step 3-6. Specify user ID for this service*

We now need to specify the user ID that the service will run under. As shown in Figure 13, you can run the service under the server's user ID, or you can specify an existing user ID that the service will run under. You cannot specify an authenticated user ID because in our example we are not protecting (for example, with basic authentication) the APIs.

## Figure 13. Deploy web service – step 6



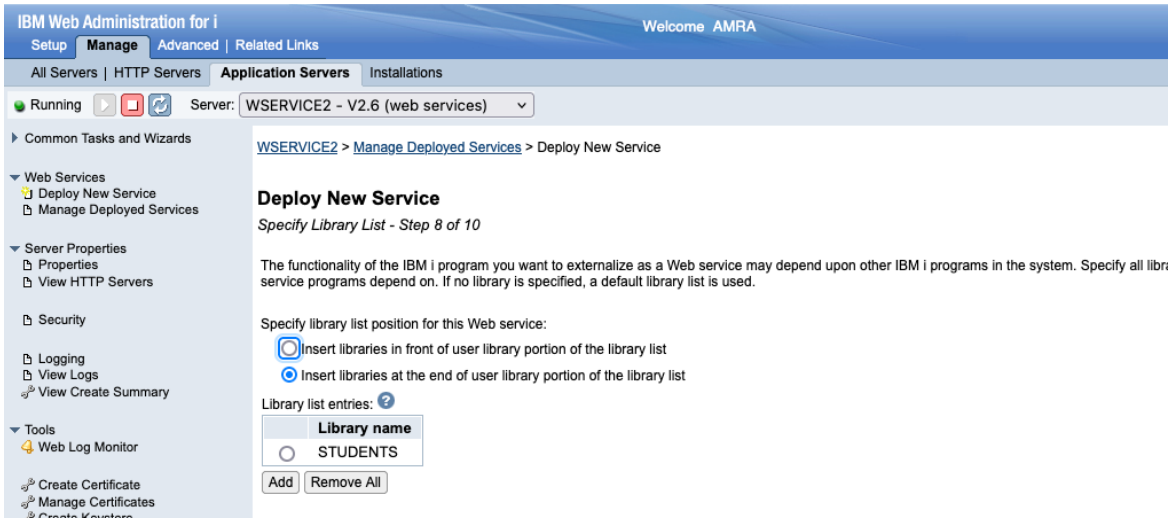

For the web service to run correctly, the user ID status must be set to *ENABLED and the password must be set to a value other than *NONE. If a user ID is specified that is disabled or has a password of *NONE, a warning message is displayed and the service may not run correctly. In addition, ensure that the specified user ID has the proper authorities to any resources and objects that the program object needs, such as libraries, databases, and files.

In this example, we will accept the default. Click on the **Next** button of the form.

### *Step 3-7. Specify library list*

Specify any libraries that the program object needs to function properly (see Figure 14).

## Figure 14. Deploy web service – step 7

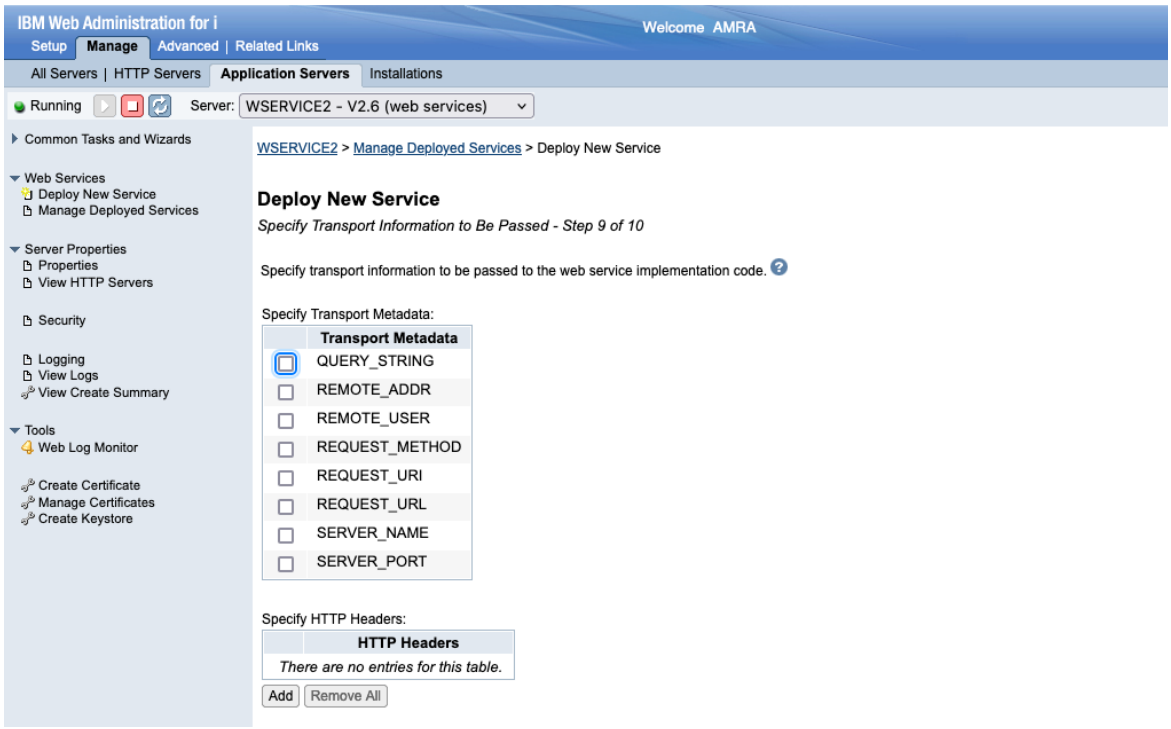



You have the option of putting the libraries at the start of the user portion of the library list or at the end of the user portion of the library list. Click on the **Next** button of the form.

## *Step 3-8. Specify transport information to be passed*

Specify what transport information related to the client request is to be passed to the Web service implementation code (see Figure 15). The information is passed as environment variables.

### Figure 15. Deploy web service – step 8




For example, the transport metadata REMOTE_ADDR is passed to web service implementation code in an environment variable named REMOTE_ADDR.

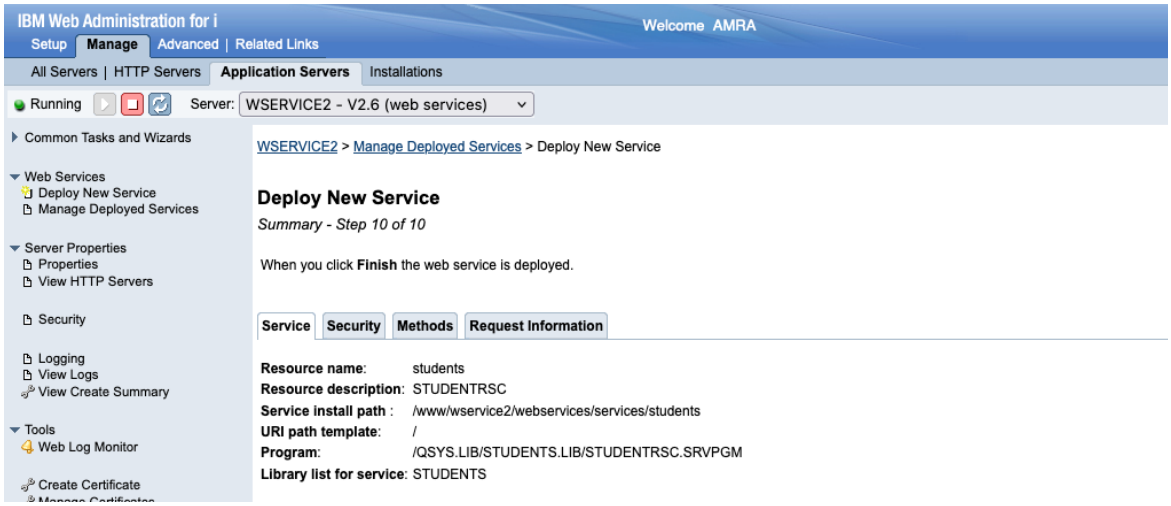
HTTP headers indicates what transport headers (e.g. HTTP headers) to pass to the web service implementation code. Transport headers are passed as environment variables. The environment variable name for HTTP headers is made up of the specified HTTP header prefixed with '`HTTP_`', all upper-cased. For example, if '`Content-type`' is specified, then the environment variable name would be '`HTTP_CONTENT-TYPE`'. If an HTTP header was not passed in on the web service request, the environment variable value will be set to the null string.

Click on the **Next** button of the form.

## *Step 3-9. Deploy web service – step 9*

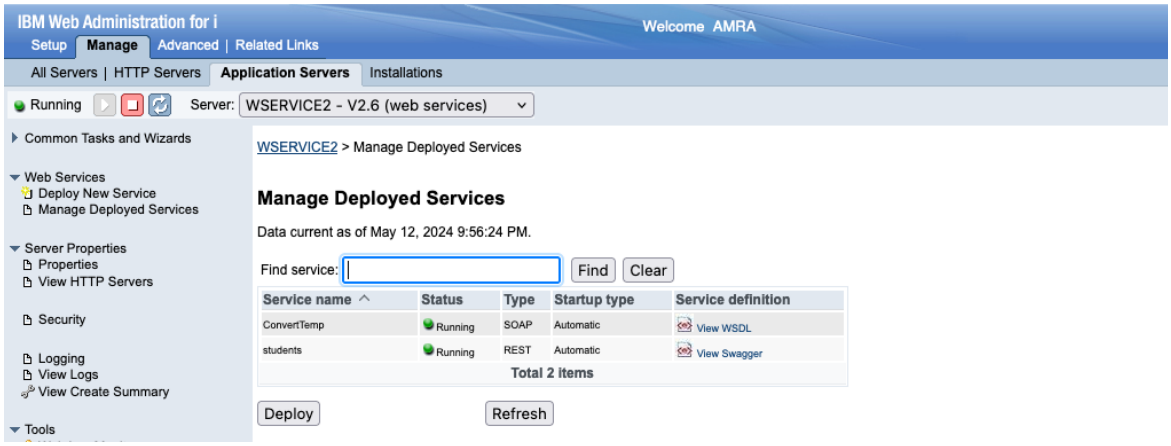
The Web service deployment wizard shows you a summary page (see Figure 16), giving you a chance to see the details relating to the Web service being deployed.

## Figure 16. Deploy web service – step 9 (Summary)




Clicking on the **Finish** button at the bottom of the summary page will kick off the installation process. When the web service is deployed the deployed service becomes active (green dot to the left of service name) as in Figure 17:

## Figure 17. Successfully deployed RESTful web service




Congratulations, you have now successfully deployed an ILE program object as a RESTful web service.

Resource methods that are bound to the HTTP GET method could be tested by using a browser. Figure 18 shows the result of the request that will return all registered students.

### Figure 18. Testing web service – return all registered students

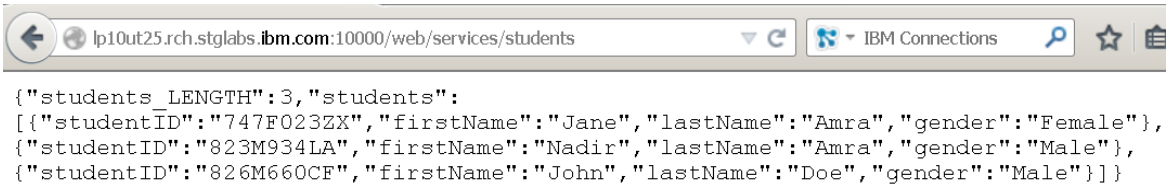
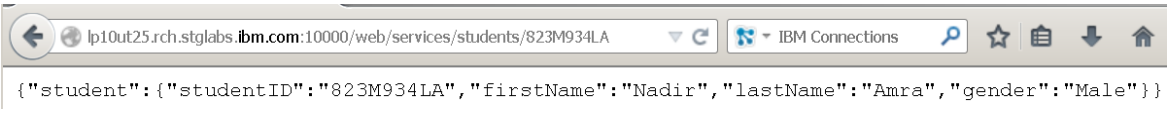


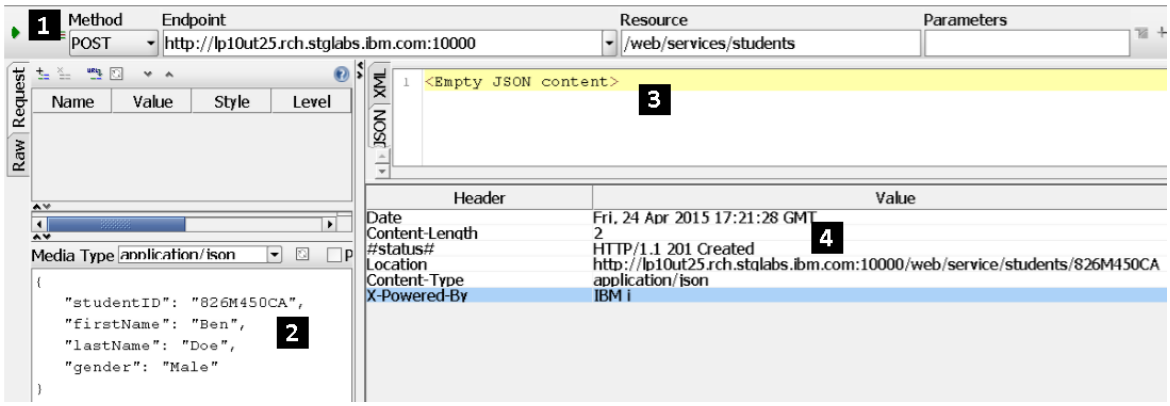

Figure 19 shows the result of a request for a student record with a student ID of 823M934LA.

### Figure 19. Testing web service – return a registered student




To test the other resource methods, an external tool needs to be used, such as SoapUI. Figure 20 shows the result of a request to create a new student registration using SoapUI.

### Figure 20. Testing web service – create a new student registration




Looking at Figure 20, since we are attempting to create a new student registration the HTTP method is POST (**1**). The sub panel numbered (**2**) is the new student registration data in JSON format that will be sent to the server as part of the HTTP POST request. After submitting the request that server response did not return any JSON data (**3**). Since the create request succeeded, the REST service returned the HTTP status code of 201 (Created) and a location header that contains the URL of the newly created student registration resource (**4**).

## Summary

In part one of this series, you learn the basic concepts behind REST web services and how the integrated web services server supports REST services. In part two of this series, you learned how to deploy a simple ILE application as a RESTful web service. In this article, you learned how to deploy a more complex ILE application that uses more of the REST features.

The integrated web services server REST support provides a solid foundation for creating and deploying REST APIs based on ILE programs or service programs on the IBM i platform. Add the highly intuitive IBM Web Administration for i GUI for deploying web services, and you've got everything you need to quickly prototype and deploy your own custom REST API.  So, what are you waiting for?

# Resources

- For everything about the integrated web services support on IBM i see the product web page.
- Parts one and two of the series can be found on the integrated web services web site.

# Code Listings

## *Source code listing for the SRA application*

```
**FREE

ctl-opt nomain pgminfo(*pcml:*module:*dclcase);

// ********************************************************************
// * LICENSE AND DISCLAIMER                                           *
// * ----------------------                                           *
// * This material contains IBM copyrighted sample programming source *
// * code ( Sample Code ).                                            *
// * IBM grants you a nonexclusive license to compile, link, execute, *
// * display, reproduce, distribute and prepare derivative works of   *
// * this Sample Code.  The Sample Code has not been thoroughly       *
// * tested under all conditions.  IBM, therefore, does not guarantee *
// * or imply its reliability, serviceability, or function. IBM       *
// * provides no program services for the Sample Code.                *
// *                                                                  *
// * All Sample Code contained herein is provided to you "AS IS"      *
// * without any warranties of any kind. THE IMPLIED WARRANTIES OF    *
// * MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND            *
// * NON-INFRINGMENT ARE EXPRESSLY DISCLAIMED.                        *
// * SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED         *
// * WARRANTIES, SO THE ABOVE EXCLUSIONS MAY NOT APPLY TO YOU.  IN NO *
// * EVENT WILL IBM BE LIABLE TO ANY PARTY FOR ANY DIRECT, INDIRECT,  *
// * SPECIAL OR OTHER CONSEQUENTIAL DAMAGES FOR ANY USE OF THE SAMPLE *
// * CODE INCLUDING, WITHOUT LIMITATION, ANY LOST PROFITS, BUSINESS   *
// * INTERRUPTION, LOSS OF PROGRAMS OR OTHER DATA ON YOUR INFORMATION *
// * HANDLING SYSTEM OR OTHERWISE, EVEN IF WE ARE EXPRESSLY ADVISED OF*
// * THE POSSIBILITY OF SUCH DAMAGES.                                *
// *                                                                  *
// *  <START_COPYRIGHT>                                               *
// *                                                                  *
// *  Licensed Materials - Property of IBM                            *
// *                                                                  *
// *  5770-SS1                                                        *
```

```
// *                                                                    *
// *  (c) Copyright IBM Corp. 2015, 2024                                *
// *  All Rights Reserved                                               *
// *                                                                    *
// *  U.S. Government Users Restricted Rights - use,                    *
// *  duplication or disclosure restricted by GSA                       *
// *  ADP Schedule Contract with IBM Corp.                              *
// *                                                                    *
// *  Status: Version 1 Release 0                                       *
// *  <END_COPYRIGHT>                                                   *
// *                                                                    *
// ********************************************************************
// *  TO CREATE SERVICE PROGRAM:                                        *
// *  (1) Database file STUDENTRSC/STUDENTDB needs to be created        *
// *      via following 2 SQL statments:                                *
// *       CREATE TABLE STUDENTS/STUDENTDB                              *
// *          ("studentID"  FOR COLUMN studentID CHAR (9) NOT NULL,     *
// *           "firstName"  FOR COLUMN firstName CHAR (50) NOT NULL,    *
// *           "lastName"   FOR COLUMN lastName  CHAR (50) NOT NULL,    *
// *           "gender" FOR COLUMN gender CHAR (10) NOT NULL,           *
// *       PRIMARY KEY ( studentID ))                                   *
// *       RCDFMT studentr                                              *
// *                                                                    *
// *                                                                    *
// *       INSERT INTO STUDENTRSC/STUDENTDB                             *
// *        (studentID, firstName, lastName, gender)                    *
// *         VALUES('823M934LA', 'Nadir', 'Amra', 'Male'),              *
// *               ('826M660CF', 'John', 'Doe', 'Male'),                *
// *               ('747F023ZX', 'Jane', 'Amra', 'Female')              *
// *                                                                    *
// *   (2)  ADDLIBLE STUDENTRSC                                         *
// *   (3)  CRTRPGMOD MODULE(STUDENTRSC/STUDENTRSC)                     *
// *                  SRCSTMF('/studentrsc.rpgle')                      *
// *   (4)  CRTSRVPGM SRVPGM(STUDENTRSC/STUDENTRSC) EXPORT(*ALL)        *
// *                                                                    *
// *                                                                    *
// ********************************************************************

DCL-F STUDENTDB DISK(*EXT) USAGE(*INPUT : *OUTPUT: *UPDATE : *DELETE)
KEYED;

// Declare HTTP status codes
DCL-C H_OK            const(200);
DCL-C H_CREATED       const(201);
DCL-C H_NOCONTENT     const(204);
DCL-C H_BADREQUEST    const(400);
DCL-C H_NOTFOUND      const(404);
DCL-C H_CONFLICT      const(409);
DCL-C H_GONE          const(410);
DCL-C H_SERVERERROR   const(500);


DCL-C ERR_DUPLICATE_WRITE const(01021);

dcl-ds studentRec qualified template;
  studentID       char(9);
  firstName       varchar(50);
```

```
  lastName        varchar(50);
  gender          char(10);
end-ds;


//*************************************************************
// Open file                                                  *
//*************************************************************
dcl-proc openStudentDB;
    dcl-pi *n int(10);
    end-pi;

    if NOT %open(STUDENTDB);
       open(e) STUDENTDB;
       if %ERROR;
         return 0;
       endif;
    endif;

    return 1;
end-proc;

//*************************************************************
// closeStudentDB                                             *
//*************************************************************
dcl-proc closeStudentDB;
    dcl-pi *n int(10);
    end-pi;

    if %open(STUDENTDB);
       close(e) STUDENTDB;
       if %error;
          return 0;
       endif;
    endif;

    return 1;
end-proc;

//*************************************************************
// getAll                                                     *
//*************************************************************
dcl-proc getAll export;
    dcl-pi *n;
      students_LENGTH   int(10);
      students          likeds(studentRec) dim(1000) options(*varsize);
      httpStatus        int(10);
      httpHeaders       char(100) dim(10);
    end-pi;

    clear httpHeaders;
    clear students;
    students_LENGTH = 0;

    openStudentDB();

    setll *loval STUDENTDB;
```

```
    read(e) studentR;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    endif;

    dow (NOT %eof);
        students_LENGTH = students_LENGTH+1;
        students(students_LENGTH).studentID =  STUDENTID;
        students(students_LENGTH).firstName =  FIRSTNAME;
        students(students_LENGTH).lastName  =  LASTNAME;
        students(students_LENGTH).gender    =  GENDER;

        read(e) studentR;
        if (%ERROR);
           httpStatus = H_SERVERERROR;
           return;
        endif;
    enddo;

    httpStatus = H_OK;
    httpHeaders(1) = 'Cache-Control: no-cache, no-store';

    closeStudentDB();
end-proc;

//*************************************************************
// getByID                                                    *
//*************************************************************
dcl-proc getByID export;
    dcl-pi *n;
      studentID             char(9);
      student               likeds(studentRec);
      httpStatus            int(10);
      httpHeaders           char(100) dim(10);
    end-pi;

    clear httpHeaders;
    clear student;

    openStudentDB();

    chain(e) studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        student.studentID = studentID;
        student.firstName = firstName;
        student.lastName  = lastName;
        student.gender    = gender;

        httpStatus = H_OK;
    else;
        httpStatus = H_NOTFOUND;
    endif;
```

```
    httpHeaders(1) = 'Cache-Control: no-cache, no-store';

    closeStudentDB();
end-proc;

//*************************************************************
// create                                                     *
//*************************************************************
dcl-proc create export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
      httpHeaders           char(100) dim(10);
    end-pi;

    openStudentDB();

    studentID = student.studentID;
    firstName = student.firstName;
    lastName  = student.lastName;
    gender    = student.gender;

    write(e) studentR;
    if NOT %ERROR;
        httpStatus = H_CREATED;
        // URL will need to change to your server and port
        httpHeaders(1) = 'Location: ' +
          'http://server:port/web/service/students/' + studentID;
    elseif %STATUS = ERR_DUPLICATE_WRITE;
        httpStatus = H_CONFLICT;
    else;
        httpStatus = H_SERVERERROR;
    endif;

    closeStudentDB();
end-proc;

//*************************************************************
// update                                                     *
//*************************************************************
dcl-proc update export;
    dcl-pi *n;
      student               likeds(studentRec);
      httpStatus            int(10);
    end-pi;

    openStudentDB();

    chain(e) student.studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        studentID = student.studentID;
        firstName = student.firstName;
        lastName  = student.lastName;
```

```
        gender    = student.gender;

        update(e) studentR;
        if NOT %ERROR;
           httpStatus = H_NOCONTENT;
        else;
           httpStatus = H_NOTFOUND;
        endif;
    else;
        httpStatus = H_NOTFOUND;
    endif;

    closeStudentDB();
end-proc;

//*************************************************************
// remove                                                     *
//*************************************************************
dcl-proc remove export;
    dcl-pi *n;
      studentID             char(9);
      httpStatus            int(10);
    end-pi;

    openStudentDB();

    chain(e) studentID STUDENTDB;
    if (%ERROR);
        httpStatus = H_SERVERERROR;
        return;
    elseif %FOUND;
        delete(e) studentR;
        if NOT %ERROR;
           httpStatus = H_NOCONTENT;
        elseif NOT %FOUND;
           httpStatus = H_NOTFOUND;
        else;
           httpStatus = H_SERVERERROR;
        endif;
    else;
        httpStatus = H_NOTFOUND;
    endif;

    closeStudentDB();
end-proc;
```