




APRIL 28, 2016

PLANNING THE DEVELOPMENT OF A Z/OS MOBILE APPLICATION TO ACCESS AUTHORIZED SERVICES

BOB ABRAMS, KIN NG, GISELA CHENG, VAUGHN PAGE, VICTOR ALONZO,
EDRIAN IRIZARRY, LUISA MARTINEZ, SUSAN DEMKOWICZ, MICHAEL SNIHUR

IBM CORPORATION
Poughkeepsie, NY 12601



Special Notices

Please Note: Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary.

Trademarks

IBM, IBM logo, LinuxONE Emperor, LinuxONE Rockhopper, and z Systems are trademarks or registered trademarks of the International Business Machines Corporation.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

IBM Mobile First is a trademark or registered trademark of the IBM Company. All statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This document is current as of the initial date of publication and may be changed by IBM at any time. Not all offerings are available in every country in which IBM operates. It is the user's responsibility to evaluate and verify the operation of any other products or programs with IBM products and programs.

THE INFORMATION IN THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT ANY WARRANTY, EXPRESS OR IMPLIED, INCLUDING WITHOUT ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND ANY WARRANTY OR CONDITION OF NON-INFRINGEMENT. IBM products are warranted according to the terms and conditions of the agreements under which they are provided.

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both, and referenced in this paper:

IBM®

RACF®

WebSphere®

z/Architecture™

IBM MobileFirst™

z/OS®

A current list of IBM trademarks is available on the web at IBM copyright and trademark information - United States (www.ibm.com/legal/us/en/copytrade.shtml).

Abstract:

Many installations today are looking to accelerate their business processes to allow immediate access to important applications, and are looking to mobile applications as a key to getting there. Mobile access improves the time it takes an information system professional (system administrator or system programmer) to obtain important information and make key decisions to maintain system availability. This white paper is intended to educate z/OS[®] mainframe professionals on building an infrastructure that provides IS personnel mobile access to z/OS system services required to do their jobs. Doing so allows installations to expose their custom services and participate in the API economy to increase the value of their System z platform assets. In particular, this whitepaper describes the design of a mobile application to invoke z/OS Runtime Diagnostics.

Introduction

There are a wealth of articles on the internet about the importance of commercial mobile applications with various opinions on the need to:

- Boost brand awareness and visibility
- Engage customers
- Provide customer service and support
- Promote your company's services
- Provide informative tips

But what about mobile applications for your own business employees? For example, applications to

- Improve employee processes within your business
- make your system staff more efficient, allowing them to access legacy z/OS applications no matter where the users are (office, meeting, hallway, coffee shop)
- Improve human reaction time to business and system problems
- Ensure that your mainframe systems continue to attain the highest level of availability possible

Have you considered defining a mobile application that allows properly-authenticated and authorized system staff to access information in and about your mainframe environment? The mainframe “continues to support modern-day tech like mobile, cloud and still-emerging technologies like the Internet of Things.”¹ You can build mobile applications that invoke back-end operating system functions and get a quick response back to the mobile device. It's the cool way to support millennial employees in doing their work!

A high percentage of system programmers carry mobile devices with them. It's a well-known fact that system programmers are on the premises 8 hours a day, but are on call 24 hours a day. We saw benefit in providing a simple mobile interface to invoke z/OS Runtime Diagnostics. z/OS Runtime Diagnostics is a z/OS function that can be invoked with an operator command when the system is experiencing degradation, or to check for potential problems. It examines the system as an experienced system operator would when a problem is occurring. Doing so saves significant time required to evaluate the system, determine what the next set of actions may be, and to identify who to assign the problem to. A mobile application to invoke this system diagnostic function can be used by the system programmer on call to check on system health while at the movies with the kids. Or the system programmer may simply need to respond about a potential system problem when not near a laptop.

¹“Mainframes are Still at the Heart of the Modern Tech World”, by Harvey Tessler. Enterprise Tech Journal, October/November 2015

Today you invoke Runtime Diagnostics using a z/OS operator command, MODIFY HZR,ANALYZE and it responds with a multiple-line message response, telling you about system problem symptoms that are indicators of a problem (if any), and what the next investigation step is. See Figure 1 for an example of the diagnostic output response. Wouldn't it be great to invoke this function from a mobile device?

We accepted the challenge to create a mobile application to invoke z/OS Runtime Diagnostics for use by a z/OS System Programmer or operations staff. Goals of the application include to ensure the identity of the user through use of RACF® or the security product of choice, invoke the Runtime Diagnostics function (requiring an authorized interface to call the operating system function), obtain the result and prepare a more modern response with all of the relevant information in the mobile response. Part of our goal was to deliver the diagnostic information in a more visually appealing, usable fashion rather than as a block of text, such as the example command response shown in Figure 1. The goal is for the user to recognize potential system problems at an instant and then act upon them immediately.

```
HZR0200I RUNTIME DIAGNOSTICS RESULT 568
SUMMARY: SUCCESS
REQ: 003 TARGET SYSTEM: SY1 HOME: SY1 2010/12/21 - 13:45:49
INTERVAL: 60 MINUTES
EVENTS:
FOUND: 02 - PRIORITIES: HIGH:02 MED:00 LOW:00
TYPES: HIGHCPU:01
TYPES: LOCK:01
-----
EVENT 01: HIGH - HIGHCPU - SYSTEM: SY1 2010/12/21 - 13:45:50
ASID CPU RATE:99% ASID:002E JOBNAME:IBMUSERX
STEPNAME:STEP1 PROCSTEP: JOBID:JOB00045 USERID:IBMUSER
JOBSTART:2010/12/21 - 11:22:51
ERROR: ADDRESS SPACE USING EXCESSIVE CPU TIME. IT MIGHT BE LOOPING.
ACTION: USE YOUR SOFTWARE MONITORS TO INVESTIGATE THE ASID.
-----
EVENT 02: HIGH - LOCK - SYSTEM: SY1 2010/12/21 - 13:45:50
HIGH LOCAL LOCK SUSPENSION RATE - ASID:000A JOBNAME:WLM
STEPNAME:WLM PROCSTEP:IEFPROC JOBID:+++++++ USERID:+++++++
JOBSTART:2010/12/21 - 11:15:08
ERROR: ADDRESS SPACE HAS HIGH LOCAL LOCK SUSPENSION RATE.
ACTION: USE YOUR SOFTWARE MONITORS TO INVESTIGATE THE ASID
```

Figure 1: Example of the multiple-line output from z/OS Runtime Diagnostics

The remainder of this white paper describes the overall architecture of the mobile application and more specifics about the various components. Topics include:

- Our choice of the mobile application integrated development environment (IDE)
- Our choice of “middleware” to receive the web request and drive its execution
- The ‘back end programming’ to invoke a native z/OS interface on behalf of the mobile application
- The transformation of the response from binary & EBCDIC to JSON
- Processing and display of the response by the mobile application

Overall architecture

For a modern mobile application to interact with back end services, the mechanism of choice is through a RESTful API.

A RESTful API is an application program interface (API) that uses HTTP requests to GET, PUT, POST and DELETE data. Representational state transfer (REST), which is used by browsers, is a programming style for a client/server interface, similar to an HTTP request, with JSON output. In order to support invocation of Runtime Diagnostics from a mobile device, we needed to define a RESTful API to invoke the internal z/OS Runtime Diagnostic function, raising several requirements that needed to be satisfied:

- A target server is required to host the RESTful API function
- Ability to securely invoke an internal z/OS authorized function
- Ability to transform the output of an internal z/OS authorized function to a form that can be consumed by the RESTful API
- Security for the RESTful API that is seamlessly integrate into the z/OS security model

After carefully considering the project requirements and comparing them to existing technology, we chose the following elements to provide an end-to-end solution, as shown in Figure 2.

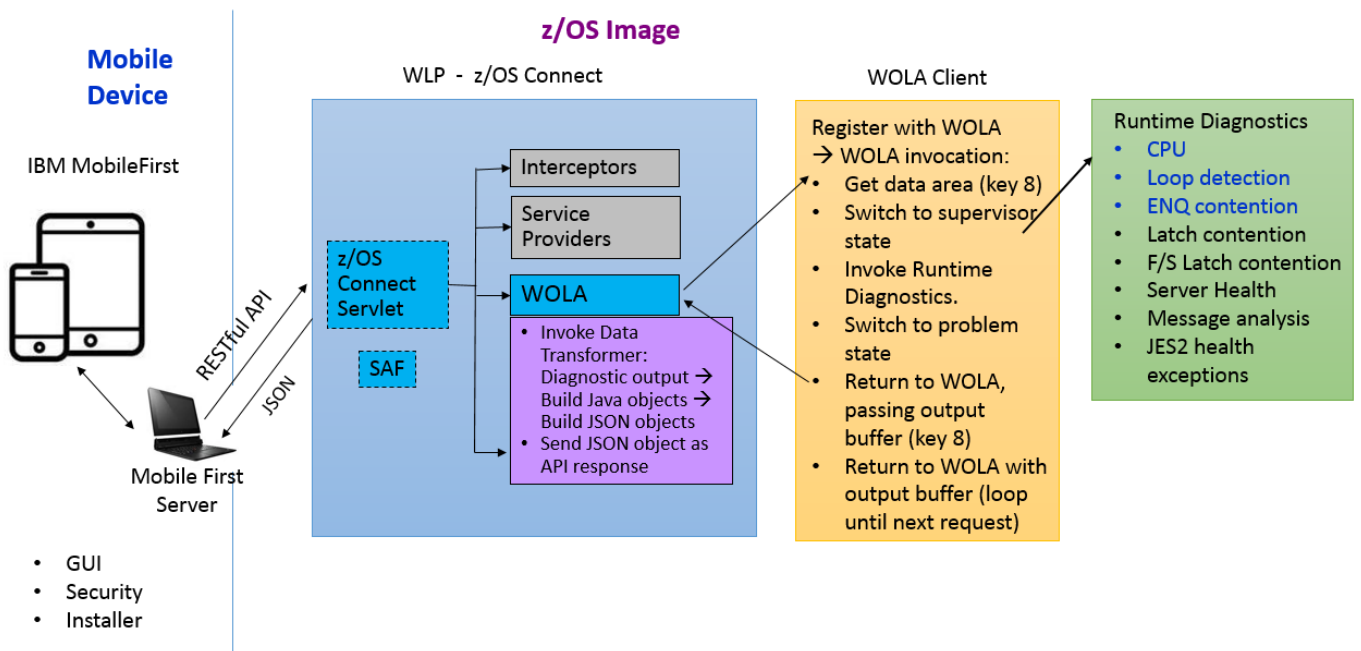


Figure 2: High level architecture of the mobile solution

Each of the following considerations are discussed in greater detail later in this paper.

- IBM MobileFirst™

We developed a mobile user interface application to invoke the z/OS function thru a new RESTful API. We required the mobile application and user interface to be supported on Android and iOS devices. This necessitated that we develop a hybrid-type mobile application, one with a common programming model that can be deployed on either platform. We selected the IBM MobileFirst product to develop the mobile application with a goal of deploying the application on Android and iOS devices.

- z/OS Connect, packaged with WebSphere® Liberty Profile (WLP)

To host the RESTful API, we chose the z/OS Connect feature of WLP. z/OS Connect simplifies the requirements needed to host a RESTful API on z/OS, as well as drive the “back end” functionality. It also allows the hosted RESTful APIs to be secured by a z/OS security product.

- Websphere Optimized Local Adapters (WOLA)

zOS Connect coordinates processing in the z/OS WebSphere environment, driving a servlet that processes the RESTful APIs coming in, invokes the security environment, and coordinates the invocation of the back end processing. The interface that provides this transition from Java to a z/OS started task (for back end processing) is WOLA. WOLA APIs are invoked by the back end program (MVS started task) to register itself as a receiver of the RESTful API, and indicate that it is ready to receive and process requests. Our back-end is a “batch WOLA client”, which is a started task written in z/OS assembler language. The started task invokes a z/OS internal program to invoke Runtime Diagnostics and coordinates its processing with zOS Connect through the WOLA APIs. WOLA APIs are also used to indirectly invoke services needed to transform z/OS internal authorized service output into a form that can be consumed by a RESTful API.

- JSON

JSON (Object JavaScript Notation) is the preferred format for data exchange through RESTful APIs. We selected the Jackson JSON packages to develop the code that is needed to transform z/OS system service output buffers into JSON format.

JSON is a text-based, human-readable data interchange format used for representing simple data structures and objects in Web browser-based code. JSON is used as an alternative to XML since JSON documents are relatively lightweight and can be processed more efficiently than XML documents.

JSON notation is used to represent the Runtime Diagnostics output, originally returned by the assembler program, to the mobile user interface “business logic”. That “business logic” processes the JSON response and directs the graphical representation of the user response.

Mobile User Interface

Since z/OS Runtime Diagnostics does not have a graphical user interface, our goal was to develop one that will support popular mobile platforms. While providing the user interface, we also portrayed the returned diagnostic report content in a way that a system programmer with little z/OS expertise would understand. Figure 3 shows the sign-in screen created for this application.



Figure 3: Runtime Diagnostics mobile sign-in screen

Next we needed to develop an application for the popular mobile platforms, including Android and iOS. Rather than develop software for each platform, the obvious choice is to use a development environment that supports both platforms and develop a single hybrid application. We developed the application, along with the response processing, using IBM's MobileFirst product so that the resulting application can be deployed on both mobile operating systems mentioned above. As an additional challenge, we experimented with ways to represent the system diagnostic relationships graphically, minimizing z/OS-specific terminology, such that a user with few years on the platform could identify the problem types. For example, contention represents a serialized relationship between two or more jobs, which could be represented graphically. Providing all of the detail lines returned by Runtime Diagnostics can easily lead to information overload. To simplify the interface and allow the information to fit on a small screen without endless scrolling, we reorganized how the diagnostic information is shown.

Of the 16 problem types supported by Runtime Diagnostics, we focused on a subset for this proof of concept - enqueue contention, loops, hangs and latch contention. In the most basic form of enqueue, a job using a particular resource is preventing (blocking) other jobs requiring that resource from completing their execution. The mobile user interface identifies this relationship with all pertinent data without propagating 6-8 lines of textual data. Figure 6 (later in this document) shows an example of the graphical display.

Figure 4 illustrates the required mobile interface sequence. The application needs to identify one or more instances of the various problem types. Subsequently, each problem type is displayed along with related information returned by the diagnostic function about each problem symptom.

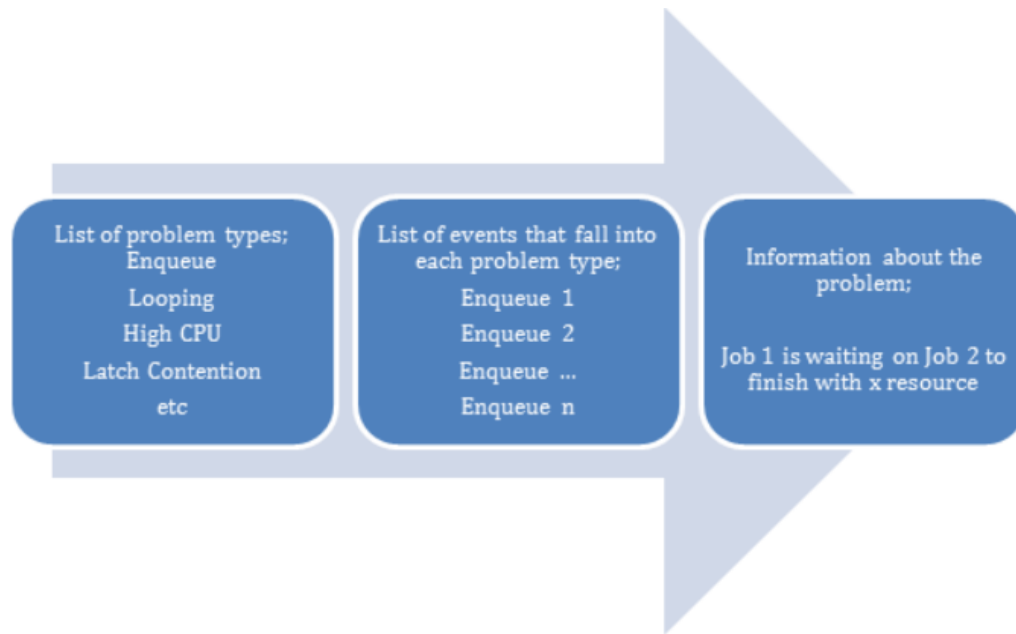


Figure 4: UI flow for three proposed screens

Communicating with the “back end”

Since this is a mobile app using IBM MobileFirst, we used an adapter to make requests to the backend. It is best to think of them as a dedicated chunk of code that handles sending and receiving the RESTful requests. To simplify the mobile application's processing of the response, the JSON stream is cleaned of "noise" and organized for easier handling. Figure 5 shows the sequence of steps taken by the mobile application to receive and process the request.

z/OS Middleware segment

The z/OS Connect feature of WLP was selected to host our RESTful API. Our selection was based on the following factors:

- z/OS Connect greatly simplifies the amount of work needed to host a RESTful API. There is no need to write a custom servlet. Defining a new RESTful API can be done by updating WLP configuration files. Writing a custom servlet would introduce complexities such as needing to invoke z/OS authorized services that do not have any Java interfaces.
- z/OS Connect's security mechanism ties into the z/OS security framework. Authorization to use the hosted RESTful APIs can be secured through the System Authorization Facility (SAF) using RACF or other security product used by the installation.

- z/OS Connect supports WOLA, which can be used to support a back end batch client that can be used to invoke z/OS internal authorized system services. WOLA supports low-latency memory-to-memory exchanges between Java applications and other functions in z/OS address spaces
- z/OS Connect supports the use of custom data transformers that can be used to transform output from z/OS services to the data formats needed by more modern programming models.

With z/OS Connect, a new RESTful API is defined by updating WLP's server.xml configuration file. As part of defining the new RESTful API, the server.xml can also be updated to:

- Specify the security model to use to check for user authentication and authorization to use the RESTful API. The RESTful API can be configured to use SAF or use a userid/password pair specified directly in the server.xml. SAF should always be used in a production environment.
- As part of defining the RESTful API in server.xml, there is also configuration work to provide a name for the service so that back end clients can register to service the RESTful API request. We will refer to these back end clients as WOLA batch clients since these are started tasks running a program that uses WOLA APIs to interact with z/OS Connect to process a RESTful API request.

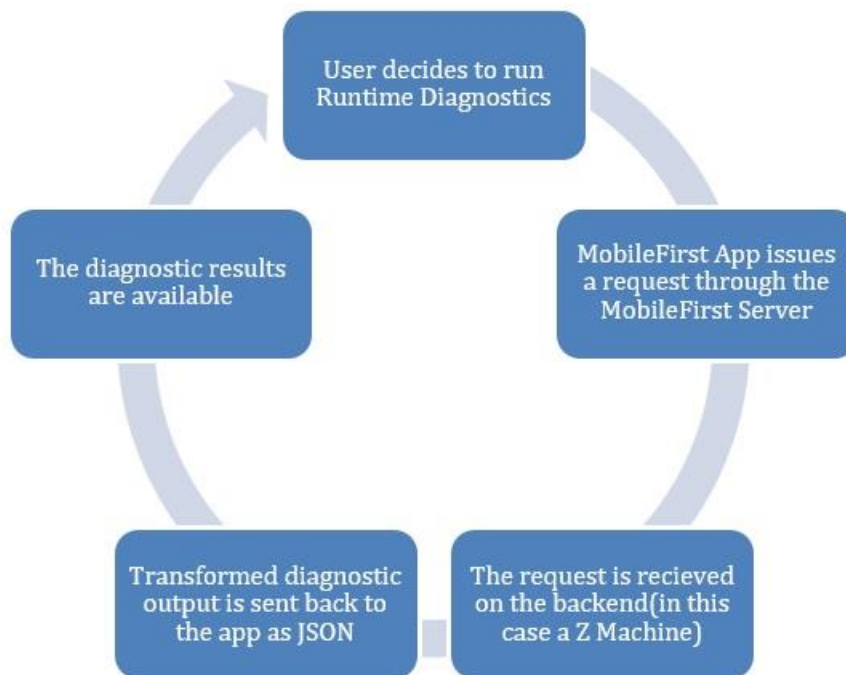


Figure 5: High level diagram of how the mobile application receives the JSON response.

- As part of defining the RESTful API in server.xml, there is also configuration work to associate a data transformer with the RESTful API. This data transformer is needed to

transform z/OS format output (EBCDIC and hexadecimal data) into a form (ASCII, JSON) that can be returned to the RESTful API caller.

WOLA Batch Client

To provide the functionality of the RESTful API, a WOLA batch client is used. A WOLA batch client is simply a z/OS assembler program that uses WOLA APIs to interact with z/OS Connect. For our implementation, we ran the WOLA batch client as a started task.

The responsibilities of the WOLA batch client are the following:

- Use a WOLA API to register as a client to service the RESTful API requests
- Use a WOLA API to wait for a RESTful API request to be received. z/OS Connect will wake the WOLA batch client when a RESTful API request is received.
- Invoke the needed z/OS authorized system service
- Use a WOLA API to return the output of the z/OS authorized system service to z/OS Connect. z/OS Connect invokes the data transformer associated with the RESTful API and pass it a Java byte array that contains the output from the z/OS authorized system service. The data transformer transforms the information contained in the Java byte array to JSON. z/OS Connect then returns the response back to the invoker of the RESTful API.
- WOLA batch client again uses the WOLA API to wait for another RESTful API request to be received

In our case, we needed to invoke a system service that requires supervisor state callers. Since the WOLA APIs only support key 8 (problem program state) programs, our batch client runs in key 8 and switches to supervisor state to invoke the authorized z/OS service and then switches back to problem program state. This also had implications on the ESTAE recovery environment established by the WOLA client to handle recovery for unexpected ABENDs.

For the prototype, we manually start the WOLA batch client after WLP has started. However, in a production environment, your system automation can start the WOLA batch clients automatically. You can also write your WOLA batch client to tolerate the scenario where WLP is not yet active, dormant until WLP initializes and recognizes the client. In addition, more than one instance of the WOLA batch client can be started. Incoming RESTful API requests will always be routed to the last WOLA batch client that is started. If a WOLA batch client terminates, incoming RESTful API requests will be routed to the active WOLA batch client that was started last. Finally, recovery should always be provided to ensure that the WOLA batch client is always available for subsequent requests.

Data Transformation

z/OS system services normally returns data in an output buffer that is mapped by an assembler mapping macro. The usual structure consists of a header section and then potentially multiple sub-sections for specific detailed data, depending on the diagnostic findings. The output buffer contains a combination of EBCDIC encoded characters and hexadecimal data.

In order for the data being returned from the z/OS system service to be consumed through RESTful APIs, the data needs to be transformed via a data transformer. In addition, a set of Java classes were required to represent the data being returned.

The responsibility of this data transformer is to perform the following:

- Traverse the output buffer that is returned by the z/OS system service
 - Convert EBCDIC encoded data into ASCII UTF equivalents
 - Convert hexadecimal data into representation that can be used by Java
 - Convert z/OS specific STCK time to Java time representation. The IBM JZOS Batch Toolkit has a utility method (`getEpochMilliseconds`) for converting STCK time into a format that is supported by Java.
- Create Java objects based on the data that is contained in the output buffer
- Transform the Java objects that were created into JSON format

In our case, the z/OS internal service returns an output buffer that contains data that has the following structure:

- A header section, with subsections for
 - summary event records
 - system event records (the number of records can vary)
 - failure event records (the number of records can vary)
 - bypass event records (the number of records can vary)

The sub-sections can be located via offsets stored in the header-subsection and traversing the subsections can be accomplished by using length fields contained within the sub-sections.

Based on the structure of our z/OS system service, we created the following Java classes:

- represent a summary event record
- represent a system event record
- represent a failure event record
- represent a bypass event record
- A main Java class was used to represent the entire output buffer of the z/OS system service. This class is mainly used as a container object for all the event record types (summary, system, failure, bypass) found in the output buffer. This is the Java object that was converted to JSON format after the data transformer has completed processing the output buffer.

Our use of the z/OS Connect feature of WLP to host our RESTful API provides the capability to use a custom data transformer for each of the RESTful API that it is hosting. In order for our data transformer to be invoked by z/OS Connect when needed, the following requirements needed to be satisfied:

- The server.xml configuration file of WLP was configured to associate our data transformer with the RESTful API
- The data transformer was written in Java and implement a SPI interface provided by z/OS Connect
- The data transformer and any dependent jars was packaged as OSGI bundles

The data transformer and dependent jar OSGI bundles needed to be packaged as a WLP feature so that it can be installed into WLP and be invoked as needed. When the back end batch client receives a request to process, it invokes the z/OS system service. When the z/OS system service returns with the output buffer, the back end batch client uses a WOLA API to respond back to the RESTful API caller. A parameter on the WOLA API to respond back to the caller points to the output buffer returned by the z/OS system service. When the WOLA API returned control to z/OS Connect, z/OS Connect had access to that output buffer. z/OS Connect then called the data transformer that is configured for that RESTful API and passed the output buffer as a Java byte array. It is then up to the data transformer code to take the byte array data and transform it to the JSON representation that the RESTful API caller expects.

Processing the JSON response – Data Reduction and Coordination

The data returned from the RESTful service is in JSON format. This makes it immediately consumable by the mobile application. It is possible to manipulate the data in a variety of ways.

We chose to convert the wordiness of the Runtime Diagnostics output into a pictorial rendering that represents the diagnostic findings. For example, we adjusted the presentation of the locking/ENQ data to represent waiter and blocker relationships. When JSON data is returned, representing the entire diagnostic response, each lock instance is represented by a complex JSON object. The objects that share the same lock are treated as individual events by Runtime Diagnostics. In cases where a single ENQ has multiple waiters, it becomes a chore for the user of the textual command output to visually interpret the relationships being reported, resorting to scanning for and remembering the names of the ENQs and creating a mental picture of what was happening. We accepted the challenge to create a graphical display for the identified problem symptoms. Figure 6 shows an example of the graphical output, where the ENQ issue is first indicated by an accordion entry. When clicked to show the details, a graphical representation is shown of the blocking job and multiple associated waiting jobs. Furthermore, the display shows the recommended next action to diagnose the impact of the illustrated contention.

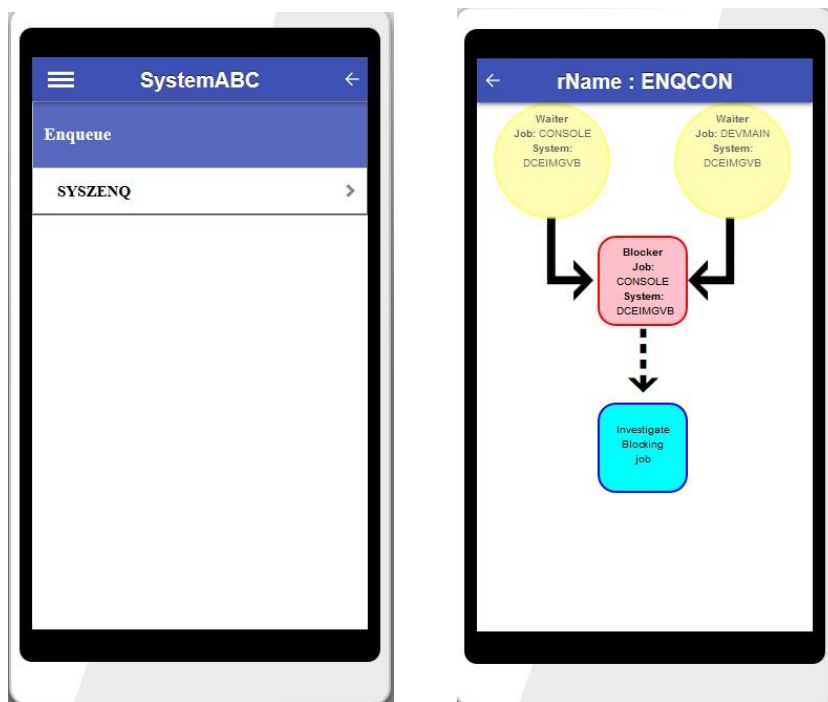


Figure 6: ENQ event list and symptom display

The code to perform this action was fairly simple. All of the diagnostic data is returned in JSON form, so it became a fairly straightforward task of looping through all of the events returned looking for duplicate ENQ names. New simpler JSON objects were then created. These objects would feature the ENQ and have a list of one or more waiters. The analysis of the data immediately at the receipt of the data from the API was performed separately from the presentation portion of the mobile application. This formed a logical structure for the mobile application.

This data reduction analysis was done in one coherent step. The isolation of function, data reduction from presentation, allowed the application to be developed independently by a group of developers. The only coordination that had to be done before the final pieces were merged was on the names of the JSON objects. This task, accomplished early on, allowed development to proceed smoothly and the final merging of the data and the presentation was also accomplished rapidly.

Resulting Mobile application

The resulting panel displays of the mobile application represent the first few screens that show the occurrence and types of problem symptoms in an accordion list style. See Figure 7 for an example of the list. Selecting an entry (row) in an accordion structure results in that that row expanding to include additional data. The accordion list allows the mobile application to display all the necessary information without having endless scrolling. For the final panel, we represent the information about the problem using directed graphs. Directed graphs allows the user to

easily identify the source of the problem and gather all the important details needed without having to parse through a long list of text.

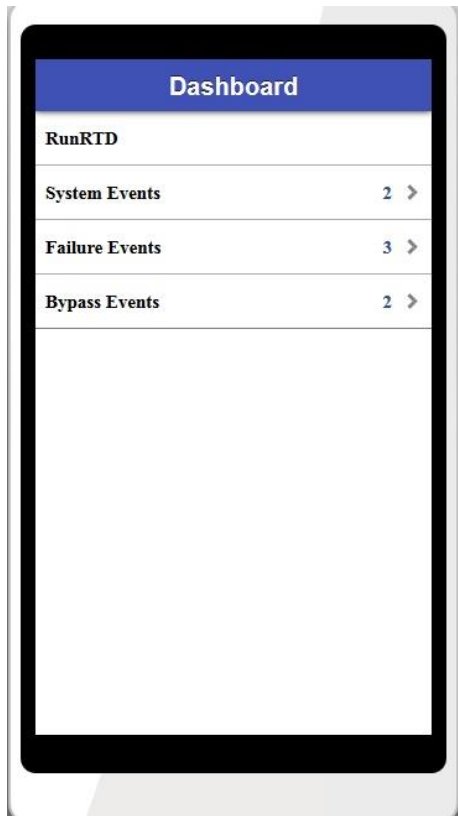


Figure 7: Mobile Application Dashboard

Conclusions

Our project demonstrated that existing z/OS assets can be exposed as a software service, without compromising security and without the need to create Java interfaces for those services. This capability can allow your enterprise to participate in the "API Economy" and let you become a "revenue generator" rather than a "cost center"².

The ability to expose your services in an easy-to-consume format can launch many new potential applications and provide opportunities to get more insight through advanced analytics. For example, our mobile application can be enhanced to support analytics (such as with Watson APIs) that may provide additional insight that may not be readily obvious.

² "The Power of the API Economy: Stimulate Innovation, Increase Productivity, Develop New Channels, and Reach New Markets", an IBM Redguide, <http://www.redbooks.ibm.com/abstracts/redp5096.html>

Using the concepts described in this paper, you too can create a web application for your business! Start small, build the architecture, and then expand the resulting application with more features to make your staff more efficient!

Our use case was z/OS Runtime Diagnostics since it's a well-defined function with potential value to z/OS system programmers. What use cases exist in your company?

- What are common z/OS inquiries your system programmers use all the time?
- What functions can you simplify for your newer (early tenure), less experienced folks?
- What do they wish they had access to?

So, instead of asking why you need a mobile application hosted on z/OS, perhaps the question is, why not?