

The IBM logo is centered in the lower half of the page. It consists of the letters "IBM" in a bold, blue, sans-serif font. Each letter is composed of eight horizontal stripes, with the top and bottom stripes being slightly longer than the others, creating a distinctive striped pattern. The background of the entire page is white, framed by a large, light blue circular arc that is thicker at the top and bottom edges, giving it a sense of depth and movement.

IBM

Contents

Key concepts.....	1
Infusing AI into IMS applications.....	4
Open access solution adoption kit.....	5
Accessing IMS data using .NET applications.....	5
Use case: Using .NET applications.....	5
Implementing open access for .NET applications.....	6
Samples overview for .NET.....	7
Accessing IMS data using Java applications.....	20
Use case: Using Java applications.....	20
Implementing open access for Java applications.....	21
Samples overview for Java.....	22
Java in IMS solution adoption kit.....	35
Use case: Java in IMS.....	35
Implementing Java in IMS.....	36
System Programmer checklist.....	37
Java Application Developer checklist.....	37
Samples overview.....	38
Sample code for starting the JMP by using Java in IMS.....	38
Sample code for input and output messages.....	40
Sample code for DFSJVMAP.....	42
Sample code for DFSJVMMS.....	43
Sample code for DFSJVMEV.....	43
Sample application code for our use case.....	44
Sample IMS Connect based client application for testing Java transactions.....	47
Sample architectural diagram.....	50
Sample z/OSMF workflow.....	51
Sample database structure.....	51
API economy solution adoption kit.....	53
Use case: API economy.....	53
Implementing API economy solution in IMS.....	54
Index.....	56

Key concepts

Ensure you are familiar with the following key concepts that are used throughout the solution adoption kit documentation.

For additional concepts, see the [IMS glossary](#).

CLASSPATH

This term is specific to Java applications. CLASSPATH indicates a list of directories and JAR files that contain resource files or Java classes that a program can load dynamically at run time. In an IMS context, the statement can be specified either in the IMS PROCLIB member DFSJVMMS (or as pointed to by the **JVMOPAS=** parameter of an IMS Java dependent region), or in a shell script that is referenced by an //STDENV DD statement.

Common Service Layer (CSL)

A collection of IMS address spaces that provide the infrastructure that is needed for advanced systems management tasks. The CSL address spaces include:

- Open Database Manager (ODBM),
- Operations Manager (OM),
- Resource Manager (RM), and
- Structured Call Interface (SCI).

The CSL is built on the Base Primitive Environment (BPE) layer.

Database description (DBD) control block

The collection of macro parameter statements that define the characteristics of a database, such as:

- the databases organization and access method,
- the segments and fields in a database record, and
- the relationship between types of segments.

Database resource adapter (DRA)

An interface to IMS DB full function databases and DEDBs. The DRA can be leveraged by an IMS coordinator controller (CCTL) or a z/OS application program that uses the ODBA interface.

Database view

In an IBM Management Console for IMS™ and DB2® for z/OS® (Management Console) context, a database view is a collection of interrelated or independent data items that are stored together to serve one or more applications.

IMS Catalog

An IMS system database used to store metadata in an XML format and that describes databases and applications characteristics. It enables solutions that require access to IMS resources metadata with IMS. Conceptually similar to the content of the IMS ACBLIB.

IMS Connect

IMS Connect is an integrated TCP/IP gateway to IMS. It provides high performance access for one or more IMS Connect clients and one or more IMS systems. IMS Connect:

- supports both IMS DB and IMS TM architecture,
- can establish direct access to IMS database through an interface called ODBM (or Open DB Manager),
- can establish a communication with the IMS TM for transaction processing support through an interface called OTMA (or Open Transaction Manager Access), and
- supports callout from IMS applications to outside services.

IMS dependent regions

An IMS dependent regions is a z/OS address space managed by IMS and used to execute applications accessing IMS resources (databases, storage, scheduling services, etc).

IMS Enterprise Suite Explorer for Development (IMS Explorer)

The IMS Enterprise Suite Explorer for Development (IMS Explorer) is an Eclipse-based graphical tool that simplifies IMS application development tasks such as updating IMS database and program definitions, and using standard SQL to manipulate IMS data.

IMS Java dependent region

The IMS Java dependent regions are IMS managed z/OS address spaces designed to accommodate the processing of java applications that need access to IMS resources. There are two types of IMS dependent regions capable of hosting Java Virtual Machine (JVM) environment:

Java message processing (JMP) regions

The JMP regions are conceptually similar to a traditional message processing program (MPP) regions, with the main difference that they allow the scheduling of Java programs only (LANG=JAVA must be specified in the IMS PSB).

Java batch processing (JBP) regions

JBP regions run flexible programs that perform batch-type processing that access IMS resources and they are conceptually similar to a traditional IMS batch processing (BMP) regions.

IMS JDBC driver

JDBC is an application programming interface (API) that IMS leverages to allow Java applications to access IMS databases. It enables application programs CRUD operations against an IMS database by using SQL queries.

IMS Universal drivers

A set of Java class libraries used by IMS to support database access using the open standard Distributed Relational Database Architecture (DRDA) specification and the distributed data management (DDM) architecture commands.

Interface

1. A shared boundary between independent systems. An interface can be a hardware component used to link two devices, a convention that supports communication between software systems, or a method for a user to communicate with the operating system, such as a keyboard.
2. A collection of operations that are used to specify a service of a class or a component.
3. In Java, a group of methods that can be implemented by several classes, regardless of where the classes are in the class hierarchy.

Java archive (JAR)

A compressed file format for storing all of the resources that are required to install and run a Java program in a single file.

Java dependent region resource adapter

IMS provides a set of Java APIs called the IMS Java dependent region resource adapter to develop Java applications to run on the IMS Java dependent regions. The IMS Java dependent region resource adapter provides Java application programs running in JMP or JBP regions with similar DL/I functionality to that provided in message processing program (MPP) and nonmessage driven BMP regions.

Java Native Interface (JNI)

A programming framework that enables Java code running in a Java Virtual Machine (JVM) to call and be called by native applications (programs specific to a hardware and operating system platform),

Java virtual machine (JVM)

An abstract software that enables a computer to run a Java program. There are three notions of the JVM:

1. specification
2. implementation
3. instance

LIBPATH

This term is specific to Java applications. In an IMS context, LIBPATH indicates the location of the library path definition for the Java environment (runtime libraries location in the z/OS UNIX System

Services (USS) environment). The LIBPATH parameter is specified either in the IMS.PROCLIB member DFSJVM EV (or as pointed to by the **ENVIRON=** parameter of an IMS Java dependent region), or a shell script that is referenced by the //STDENV DD statement.

Management Console for IMS and DB2 for z/OS (Management Console)

An application server that consolidates information about your IMS and DB2 resources on z/OS into a single, intuitive, graphical web interface. This interface is used on a client to connect to different services on z/OS. Management Console can accelerate your analysis of your z/OS environment and reduce the need for advanced mainframe skills.

Maven

This is a specific term related to Java applications. Maven is a build automation tool used primarily for Java projects. Maven addresses two aspects of building software: first, it describes how software is built, and second, it describes its dependencies.

Open Database Manager (ODBM)

It provides an environment that manages access to online IMS databases from anywhere in the enterprise. It supports open-standards for connectivity to IMS databases and leverage the so called Common Service Layer (CSL) address spaces.

Operations Manager (OM)

In an IMSplex environment, OM provides a more flexible way to issue a command against multiple IMS systems in a plex. Its application programming interface (API) can also be used for automated operator programs (AOPs). OM receives commands from AOPs, routes the command to IMSplex members, consolidates commands responses, and sends the responses to the AOP, embedded in XML tags. Part of the so called IMS Common Service Layer (CSL) component.

Program communication block (PCB)

A control block that contains pointers to Information Management System (IMS) databases.

Program specification block (PSB)

The PSB describes the way a database is viewed by an IMS application program. Specifically a PSB declares the database segments an application program can access along with the functions it can perform on the data such as read, update or delete. PSBs are composed of one or more Program Communication Block (PCB), one for each DB accessed. Whenever a program needs to access a new segment on an existing database or whenever it needs to access a new database, the PSB must be altered.

PROCLIB data set

An authorized z/OS partitioned data set that contains a collection of procedures (members) used indirectly by IMS components, IMS services, or by users needing to activate pre-determined IMS resources.

Resource Manager (RM)

A Common Service Layer (CSL) component that is used by IMS to manage resources and coordinates online change for IMS systems in an IMSplex.

Structured Call Interface (SCI)

A CSL component that manages communications between the members belonging to the same IMSplex (IMS systems).

Segment

In an IMS database context, it is the unit of access to a database; for the database system, the smallest amount of data that can be transferred by one IMS operation.

Superuser

In a UNIX context, a superuser is a UNIX system administrator with authority above and beyond that of the ordinary user. A superuser, for example, can:

- stop any runaway program,
- purge corrupted files,
- change file or directory permissions,
- reset passwords when users forget them,
- remove users' permission to use the system, and etc...

In a UNIX System Services (USS) context, superusers are defined in RACF (or equivalent). SUPERUSER state (when so defined in RACF) can be entered by the command **SU** in the USS Shell.

Symbolic link

A type of file that contains a pointer to another file or directory.

UNIX System Services (USS)

USS is a component of z/OS. It is a certified UNIX operating system implementation (XPG4 UNIX 95) optimized for mainframe architecture. Also referred to as z/OS UNIX, it supports a hierarchical file system familiar to the typical UNIX users.

Infusing AI into IMS applications

Applications that run in IMS can make more timely and better decisions, and achieve improved business outcomes by capitalizing on AI within their transactions.

What is AI and why use it on IBM zSystems?

Artificial intelligence (AI) is broadly used in the technology world to describe solutions that can learn on their own. Machine learning (ML) is a subset of AI and encompasses algorithms that make predictions by applying statistical methodologies to identify patterns in past behavior. Deep learning (DL), which is a subset of machine learning, uses neural networks that can, when exposed to different situations or patterns of data, learn on their own. Deep learning refers to a neural network that consists of more than three layers, including the input and the output. You can learn about typical use cases with AI on IBM zSystems at [Journey to AI on IBM Z and LinuxONE](#).

IBM zSystems, and the IBM Integrated Accelerator for AI incorporated in IBM z16, can optimize the processing of machine learning and deep learning algorithms. You can take advantage of these capabilities when using suitable AI models with any supported release of IMS.

Why infuse AI in IMS applications?

Infusing AI is about being able to apply AI across your enterprise, drawing on predictions, automation, and optimization to improve your business decisions and outcomes. It's also about making AI part of your day-to-day operations.

A financial application running in IMS can use AI to secure transactions with external applications. In a scenario like claims fraud prevention, TM could call an AI model in Watson Machine Learning for z/OS (WMLz) to determine the outcome of a transaction based on a prediction of whether it is likely to be fraudulent.

By infusing AI models into applications running in IMS, you enable real-time decision making within the transactions, significantly reduce latency over making calls off-platform, and avoid the need for the data to leave the platform. The data that provides input to AI models is often relevant only at the time the transaction is being processed: Is there enough funds to process this transaction? Are there any signs of fraud? Can this transaction be fraudulent?

How do I infuse AI in my IMS applications?

There are a variety of methods for infusing AI in your IMS applications. The choice of which option works best for your use case depends on a number of considerations, including the selection of AI model and where it is to be deployed, the response time required by your transaction, and the tools and products that are already in use within your enterprise.

Some of the most common methods for infusing AI into IMS applications are:

IBM Operational Decision Manager (ODM) with WMLz:

You can invoke an enhanced ODM rule from the application to reference a model deployed to WMLz and use the prediction from the model in the rule.

IBM Watson Machine Learning on z/OS (WMLz):

You can make the REST API call from the application to invoke a model in WMLz running on z/OS.

A community-available AI framework, such as IBM Snap Machine Learning (Snap ML), TensorFlow, or PyTorch:

You can make a REST call from the application to an AI model deployed to the AI framework that is hosted either in an IBM z/OS Container Extension (zCX) within the z/OS environment, or in Linux on IBM Z.

For more information about each option and how to choose among them, see the [Learn more](#) section of [Journey to AI on IBM Z and LinuxONE](#).

Open access solution adoption kit

Historically, IMS data residing on IBM z Systems was only accessible by COBOL and PL/I applications on the mainframe. With the open access solution, you can now access IMS database from outside the IBM z Systems platform. Accessing IMS data outside of the IBM z Systems platform is called *open access*.

Related information

[Getting started with open access](#)

Accessing IMS data using .NET applications

If you have .NET applications outside of IMS that need access to the data, whether it is for querying or analytics or any purpose, the open access kit can guide you to make that happen.

Use case: Using .NET applications

The following use case depicts a fictitious Insurance company as it navigates through and successfully implements the open access solution.

Goal:

Prosentient Insurance, a fictitious company, intends to develop a new Claim Handling client portal, which requires that IMS data be easily accessible (open) to the .NET apps that will comprise the new portal.

Preconditions:

- Data is located across multiple platforms
- A lack of integration between key systems has contributed to the current sluggish client experience
- Prosentient has more .NET skills in house than COBOL skills

Success End Condition:

.NET Developers can easily create and deploy apps that access IMS data for the new portal, without requiring deep knowledge of the hierarchical data model or IMS's DL/I access language.

Primary Actors:

- Vikram, System Programmer at Prosentient
- Susan, Application Developer at Prosentient
- Steve, Database Administrator at Prosentient

Secondary Actors:

- Ana, Enterprise Architect at Prosentient

- Lars, an independent industry consultant to Prosentient
- Matt, Project Manager at Prosentient
- Trish, SAF Administrator at Prosentient

Main Use Case Success Scenario:

1. Each of the people involved in the project familiarize themselves with the overall structure of the open access architecture, with particular focus on the areas for which they have responsibility.
2. Vikram sets up the Common Service Layer infrastructure required for open access to IMS data.
3. Vikram sets up the IMS Catalog
4. Steve downloads and installs the IMS Explorer
5. Steve captures the metadata for IMS databases using IMS Explorer
6. Susan verifies the IMS database schema using IMS Explorer. As an application developer, she would most likely be familiar with all of the application databases.
7. Susan creates and executes SQL queries using IMS Explorer
8. Susan deploys IMS database resource adapters in WebSphere Liberty Server
9. Susan develops a .NET application that uses the SQL queries to access IMS data

Related information

[Getting started with open access](#)

Implementing open access for .NET applications

To get started with open access to IMS data, the first step is to set up the open access infrastructure to provide access to .NET applications. Then, you can develop applications that contains standard ADO.NET programming calls to execute SQL queries.

About this task

It is important to understand that although SQL syntax is used, under the covers, IMS translates SQL to DL/I and vice verse.

<i>Table 1. Implementing open access</i>	
Procedure	User role who completes the task
1. Learn and get started with open access. 1. “Use case: Using .NET applications” on page 5 (topic) 2. Getting Started with opening access to IMS data (video) 3. Introduction to the IMS Net Data Provider (video)	Everyone on the team, to get an understanding of what is involved

Table 1. Implementing open access (continued)	
Procedure	User role who completes the task
2. Set up the infrastructure needed for open access to IMS database. 1. Setting up open access infrastructure (article) <ul style="list-style-type: none"> • “Sample code for getting started with open access” on page 10 (topic, sample code) 2. Setting up the IMS catalog (article) <ul style="list-style-type: none"> • “Sample code for setting up the IMS Catalog” on page 14 (topic, sample code) 3. “Sample .NET architectural diagram” on page 10 (topic, sample code)	Vikram, the System Programmer
3. Install IMS Explorer for Development (topic)	Steve, the DBA and Susan, the Application Developer
4. Explore, Visualize and Understand IMS database schema (video) <ul style="list-style-type: none"> • “Sample database structure” on page 18 (topic, sample code) 	Susan, the Application Developer
5. Creating and executing SQL Queries with IMS Explorer (video)	Susan, the Application Developer
6. Developing a console application using IMS NET Data Provider (video)	Susan, the Application Developer
7. Developing an ASP NET web application using IMS NET Data Provider (video) <ul style="list-style-type: none"> • “Sample .NET application code for our use case” on page 7 (topic, sample code) 	Susan, the Application Developer

Samples overview for .NET

The following sample code pertains to the use case scenario for the insurance company. Use these .NET samples to understand how to implement open access and how to develop the application that queries the database.

Sample .NET application code for our use case

This sample application retrieves customer information about insurance policies for the given customer number provided.

The application gets a connection to the IMS insurance database from app.config using ConfigurationManager. This connection is then used to access IMS data using SQL calls.

```

using System;
using System.Configuration;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics;
using System.IO;
using System.Data;
using IBM.Data.IMS;

namespace IMS_Data_Provider_Example
{
    class CustomerInfoService
    {
        static FileStream objStream;

        static void Main(string[] args)
        {
            // Configure log
            objStream = new FileStream("C:\\\\CustomerInfoServiceLog.txt", FileMode.Create);
            TextWriterTraceListener objTraceListener = new TextWriterTraceListener(objStream);
            Trace.Listeners.Add(objTraceListener);
            Trace.AutoFlush = true;

            int num = Int32.Parse(args[0]);
            GetCustomerInfo(num);

            Trace.Flush();
            Trace.Listeners.Remove(objTraceListener);
            objStream.Close();
        }

        static void GetCustomerInfo(int customerNumber)
        {
           ConnectionStringSettings cfg = ConfigurationManager.ConnectionStrings["INSURANCEDB"];

            using (IMSConnection conn = new IMSConnection(cfg.ConnectionString))
            {
                try
                {
                    conn.Open();

                    IMSCommand customerStatement = new IMSCommand("SELECT FIRSTNAME, LASTNAME,
CUST_STREET, CUST_CITY, CUST_STATE, CUST_ZIPCODE, DATEOFBIRTH FROM INSURPCB.CUSTOMER WHERE
CUSTOMERNUMBER = ?", conn);
                    IMSCommand autoPolicyStatement = new IMSCommand("SELECT POLICYNUMBER,
CAR_MAKE, CAR_MODEL, CAR_MANUFACTUREDATE, CAR_REGNUMBER, CAR_DRIVERNAME FROM INSURPCB.POLICY
WHERE CUSTOMER_CUSTNO = ? and POLICYTYPE='A'", conn);
                    IMSCommand housePolicyStatement = new IMSCommand("SELECT POLICYNUMBER,
HOMEPROPERTYTYPE, HOMEBEDROOMS, HOMEHOUSEVALUE, HOME_STREET, HOME_ZIPCODE FROM INSURPCB.POLICY
WHERE CUSTOMER_CUSTNO = ? and POLICYTYPE='H'", conn);

                    IMSParameter custNumber = new IMSParameter("CN", IMSType.INTEGER);
                    custNumber.Value = customerNumber;
                    customerStatement.Parameters.Add(custNumber);
                    autoPolicyStatement.Parameters.Add(custNumber);
                    housePolicyStatement.Parameters.Add(custNumber);
                }
            }
        }
    }
}

```

```

using (IMSDDataReader customerRS = customerStatement.ExecuteReader())
{
    if (customerRS.Read())
    {
        Console.WriteLine("First Name: " + customerRS.GetString(0));
        Console.WriteLine("Last Name: " + customerRS.GetString(1));
        Console.WriteLine("Street: " + customerRS.GetString(2));
        Console.WriteLine("City: " + customerRS.GetString(3));
        Console.WriteLine("State: " + customerRS.GetString(4));
        Console.WriteLine("Zip: " + customerRS.GetString(5));
        Console.WriteLine("Date of Birth: " + customerRS.GetString(6));
        customerRS.Close();

        using (IMSDDataReader houseRS = customerStatement.ExecuteReader())
        {
            while(houseRS.Read())
            {
                Console.WriteLine("Policy Number: " + houseRS.GetInt32(0));
                Console.WriteLine("Property type: " + houseRS.GetString(1));
                Console.WriteLine("Number of bedrooms: " +
houseRS.GetInt16(2));

                Console.WriteLine("Value: " + houseRS.GetInt32(3));
                Console.WriteLine("Street: " + houseRS.GetString(4));
                Console.WriteLine("Zip: " + houseRS.GetString(5));
            }
            houseRS.Close();
        }

        using (IMSDDataReader autoRS = customerStatement.ExecuteReader())
        {
            while(autoRS.Read())
            {
                Console.WriteLine("Policy Number: " + autoRS.GetInt32(0));
                Console.WriteLine("Make: " + autoRS.GetString(1));
                Console.WriteLine("Model: " + autoRS.GetString(2));
                Console.WriteLine("Manufacture Date: " + autoRS.GetString(3));
                Console.WriteLine("Registration Number: " +
autoRS.GetString(4));

                Console.WriteLine("Driver Name: " + autoRS.GetString(5));
            }
            autoRS.Close();
        }
    }
}
conn.Close();
}

```

```

catch (IMSEException ex)
{
    if (conn.State == ConnectionState.Open)
        conn.Close();

    Console.WriteLine(ex.Message);
    throw ex;
}
catch (Exception ex)
{
    if (conn.State == ConnectionState.Open)
        conn.Close();

    Console.WriteLine(ex.Message);
    throw ex;
}
}
}
}

```

Sample .NET architectural diagram

The following figure shows ADO.NET application accessing IMS database using IMS .NET Data Provider.

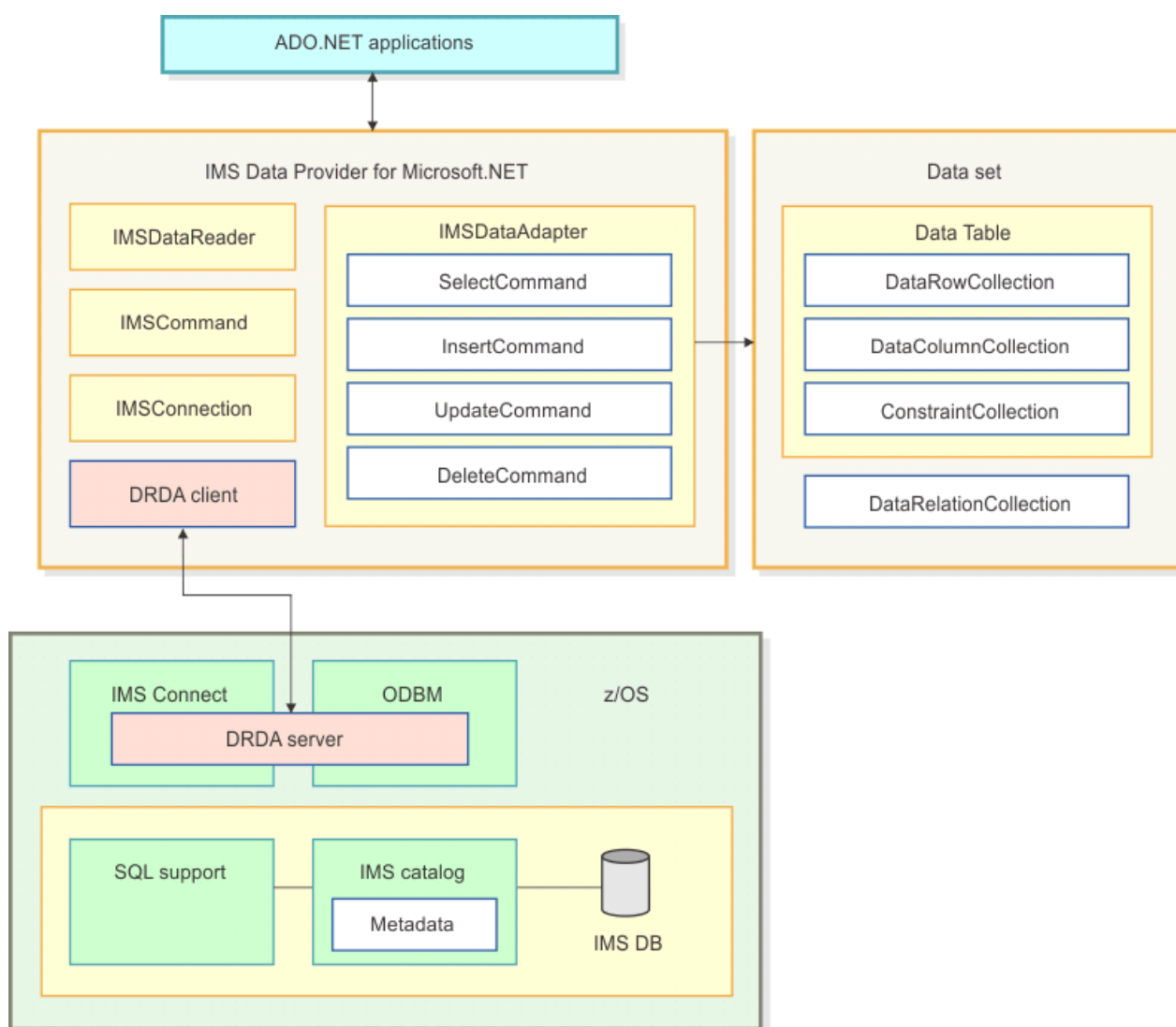


Figure 1. ADO.NET application that is using IMS .NET Data Provider to access IMS

Sample code for getting started with open access

This code shows sample configurations for procedures, configuration members and JCL for setting up an IMS system to provide open access.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

IMS Connect JCL and PROCLIB member

IMS Connect JCL and PROCLIB member contain the following sample code:

HWSCFG00 PROCLIB member: Below are sample IMS Connect Configuration parameters. The ODACCESS statement specifies parameters related to IMS open access. Here, the port number is 5555, port timeout value is set to 60 seconds, ODBM timeout is also set to 60 secs. The IMSplex name is P1ex1 and the IMS Connect name within that IMSplex is IVP14HWS. The **ODBMAUTOCONN=Y** parameter specifies that IMS Connect will automatically connect to new and existing instances of ODBM within that IMSplex.

```

HWS=(ID=IVP14HWS,XIBAREA=100,RACF=N,RRS=Y)
TCPIP=(HOSTNAME=TCPIP,PORTID=(9999,LOCAL),RACFID=RACFUID1,TIMEOUT=5000)
DATASTORE=(GROUP=IVPXCFCN,ID=IVP1,MEMBER=IVP14HWS,DRU=HWSYDRU0,
TMEMBER=IVP1)
ODACCESS=(ODBMAUTOCONN=Y,IMSPLEX=(MEMBER=IVP14HWS,TEMBER=PLEX1),
DRDAPORT=(ID=5555,PORTMOT=6000),ODBMTMOT=6000)

```

IMS Connect Startup PROCLIB: Below is sample JCL to start the IMS Connect address space.

Note: We are using **HWSCFG00** for the IMS Connect configuration parameters and **BPECFGHT** for the BPE configuration parameters.

```

//HWS1      PROC  RGN=0M,SOUT=A,
//          BPECFG=BPECFGHT,
//          HWSCFG=HWSCFG00
// *
// *****
// * BRING UP AN IMS CONNECT USING BPEINI00
// *****
//STEP1     EXEC  PGM=BPEINI00,REGION=0M,TIME=1440,
//              PARM='BPECFG=&BPECFG,BPEINIT=HWSINI00,
//              HWSCFG=&HWSCFG'
//STEPLIB DD  DSN= IMS.SDFSRESL,DISP=SHR
//PROCLIB DD  DSN= IMS.PROCLIB,DISP=SHR
//SYSPRINT DD  SYSOUT=&SOUT
//SYSUDUMP DD  SYSOUT=&SOUT

```

IMS PROCLIB member for IMSplex

IMS PROCLIB member for IMSplex contains the following sample code:

DFSCGxxx PROCLIB member: The DFSCGxxx PROCLIB member contains various Common Service Layer (CSL) parameters. In this IMS open access solution adoption kit, there are two parameters needed:

1. **IMSPLEX=plex1** specifies the name of our IMSplex as plex1.
2. **RMENV=N** specifies that the Resource Manager (RM) component of CSL is not to be used.

The suffix for DFSCGxxx, xxx, is specified on the **CSLG=** parameter in the IMSPBxxx PROCLIB member.

```

IMSPLEX=plex1,
RMENV=N

```

ODBM JCL and PROCLIB members

ODBM JCL and PROCLIB members contain the following sample code:

BPEODBM PROCLIB member: Below is sample BPE configuration PROCLIB member for ODBM. In this member, the internal dispatcher, AWE, CSL and ODBM traces are set to HIGH and all other internal traces are set to Low. These traces are typically examined by the IMS support team for problem analysis.

```

# BPE CONFIGURATION MEMBER for ODBM - BPEODBM
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE)                /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE          */
TRCLEV=(AWE,HIGH,BPE)             /* AWE SERVER TRACE          */
# ODBM Traces
TRCLEV=(*,LOW,ODBM)
TRCLEV=(CSL,HIGH,ODBM)
TRCLEV=(ODBM,HIGH,ODBM)

```

CSLDC001 PROCLIB member: Below are sample parameters for the CSLDCxxx PROCLIB member. In the **LOCAL_DATASTORE_CONFIGURATION** section, ODBM (with the name of OD1) will connect to an IMS datastore with an ID of IMS1. The IMS datastore is IMS1, and the ALIASes used by the applications are: IM1A and IM1B.

```

*****
* CSLDC001 ODBM CSL PROCLIB MEMBER *
*****
<SECTION=GLOBAL_DATASTORE_CONFIGURATION>

IDRETRY=5                /* Retry connection 5 times before quit */
MAXTHRS=50               /* 10 threads max to any IMS Datastore */
TIMER=30                 /* 30 seconds between ID retry attempts */
FPBUF=10                 /* 10 DEDB buffers per thread */
FPBOF=10                 /* 10 Overflow buffers per thread */
CNBA=200                 /* (FPBUF+FPBOF)*MAXTHRS <= CNBA */
*****
* Define DATASTORE properties for ODBM name OD1 *
*****
<SECTION=LOCAL_DATASTORE_CONFIGURATION>
ODBM(NAME=OD1,           /* Define parms for OD1 */
  DATASTORE(NAME=IMS1,  /* IMSID on LPAR A */
    ALIAS(NAME=IM1A,NAME=IM1B), /* Names for APPL sets 1 & 2 */
  )
)

```

CSLDI001 PROCLIB member: Below is sample CSLDIxxx PROCLIB member. In this member, the **IMSPLEX(NAME=PLEX1)** parameter specifies the IMSplex name (PLEX1) that ODBM participates in. The **ODBMCFG=001** parameter specifies the suffix for the CSLDCxxx PROCLIB member. The **ODBMNAME=OD1** parameter specifies OD1 as the name of this ODBM.

```

*-----*
* ODBM INITIALIZATION PROCLIB MEMBER - CSLDI001 *
*-----*
ARMRST=N,                /* SHOULD ARM RESTART OM ON FAILURE */
IMSPLEX(NAME=PLEX1)      /* IMSPLEX NAME (CSLPLEX1) */
ODBMCFG=001              /* suffix for CSLDCxxx (CSLDC001) */
ODBMNAME=OD1,            /* ODBM NAME (ODBMID = OD10D) */
RRS=Y                    /* RRS=Y will use ODBA access */
*-----*

```

ODBM Startup PROC: Below is sample JCL to startup the ODBM address space.

Note: Below is a sample JCL procedure to start the ODBM address space. The PROCLIB member BPEODBM is being used for the BPE configuration parameters, and the ODBMINIT=001 parameter indicates the PROCLIB member CSLDI001 is being used for the ODBM initialization parameters.

```

/*-----*/
/* ODBM Startup Proc */
/*-----*/
//IMS10DBM PROC RGN=3000K,SOUT=X,
//                BPECFG=BPEODBM,
//                ODBMINIT=001
//*
//IMS10DBM EXEC PGM=BPEINI00,REGION=&RGN,
//  PARM=('BPECFG=&BPECFG','BPEINIT=CSLDINI0','ODBMINIT=&ODBMINIT')
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT

```

OM JCL and PROCLIB members

OM JCL and PROCLIB members contain the following sample code:

BPECONFG PROCLIB member: Below is a sample BPE configuration PROCLIB member that is used by the OM address space.

```
# BPE CONFIG MEMBER for SCI & OM - BPECONFG
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE)          /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE */
TRCLEV=(AWE,HIGH,BPE)       /* AWE SERVER TRACE */
```

CSLOI000 PROCLIB member: Below is a sample CSLOIxxx PROCLIB member that contains the OM initialization parameters.

```
*-----*
* OM INITIALIZATION PROCLIB MEMBER - CSLOI000 *
*-----*
ARMRST=N,          /* SHOULD ARM RESTART OM ON FAILURE */
CMDLANG=ENU,       /* USE ENGLISH FOR COMMAND DESC */
CMDSEC=N,          /* NO COMMAND SECURITY */
OMNAME=OM1,        /* OM NAME (OMID = OM1OM) */
IMSPLEX(
  NAME=PLEX1,       /* IMSPLEX NAME (CSLPLEX1) */
  AUDITLOG=SYSLOG.OM2Q01.LOG), /* MVS LOG STREAM */
CMDTEXTDSN=SSTBLD.I14APAR.DBDC.M.CM1.SDFSDATA /* CMD TXT DATAS */
*-----*
```

OM Startup PROC: Below is a sample JCL procedure to start the OM address space. It uses the BPECONFG PROCLIB member for the BPE configuration parameters and **OMINIT=000** specifies the suffix for the CSLOIxxx PROCLIB member.

```
//*-----*/
/* OM Startup Proc */
/*-----*/
//IMS10M PROC RGN=3000K,SOUT=X,
//          BPECFG=BPECONFG,
//          OMINIT=000
//
/*
//IMS10M EXEC PGM=BPEINI00,REGION=&RGN,
// PARM=('BPECFG=&BPECFG','BPEINIT=CSLOINI0','OMINIT=&OMINIT')
/*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
/*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
```

SCI JCL and PROCLIB members

SCI JCL and PROCLIB members provide the following sample code:

BPECONFG PROCLIB member: Below is a sample BPE configuration PROCLIB member that is used by the SCI address space.

Note: This is the same member that is used by the OM address space.

```
# BPE CONFIG MEMBER for SCI & OM - BPECONFG
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE)          /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE */
TRCLEV=(AWE,HIGH,BPE)       /* AWE SERVER TRACE */
```

CSLSI000 PROCLIB member: Below is sample CSLSIxxx PROCLIB member that contains the SCI initialization parameters. The **IMSPLEX(NAME=PLEX1)** parameter specifies the IMSplex name that SCI participates in. The **SCINAME=SCI1** parameter specifies the SCI address space name.

```

*-----*
* SCI INITIALIZATION PROCLIB MEMBER - CSLSI000 *
*-----*
ARMRST=N, /* SHOULD ARM RESTART SCI ON FAILURE */
IMSPLEX(NAME=PLEX1) /* IMSPLEX NAME (CSLPLEX1) */
SCINAME=SCI1, /* SCI NAME (SCIID = SCI1SCI) */

```

SCI Startup PROC: Below is a sample JCL procedure to start the SCI address space. It uses the BPECONFIG PROCLIB member for the BPE configuration parameters, and **SCIINIT=000** specifies the suffix for the CSLSIxxx PROCLIB member, which in this case is pointing to CSLSI000.

```

/*-----*/
/* SCI Startup Proc */
/*-----*/
//IMS1SCI PROC RGN=3000K,SOUT=X,
// BPECFG=BPECFG,
// SCIINIT=000
/*
//IMS1SCI EXEC PGM=BPEINI00,REGION=&RGN,
// PARM=('BPECFG=&BPECFG','BPEINIT=CSLSINI0','SCIINIT=&SCIINIT')
/*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
/*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT

```

Sample code for setting up the IMS Catalog

These sample configurations for procedures, configuration members and JCL show how to set up the IMS catalog.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

DFS3UACB

Below is a sample JCL for running the DFS3UACB (ACB Generation and Catalog Populate utility) as a BMP job. You can run DFS3UACB as a BMP when the IMS catalog is being used by the IMS online system. Ensure that your IMS system has the catalog database open for update; the default is read-only. For more information about this utility, see [ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Utilities\)](#).


```

//ACBPOPUP JOB 'IMS SYSTEM',CLASS=K,MSGLEVEL=(1,1),REGION=0M
//*
//*****
//* DFS3UACB GENERATES ACB MEMBERS IN AN ACB LIBRARY BY CALLING THE
//* ACB MAINTENANCE UTILITY. IN THE SAME JOB STEP,
//* DFS3UACB INSERTS RECORDS IN THE EXISTING IMS CATALOG BY CALLING
//* THE IMS CATALOG POPULATE UTILITY (DFS3PU00)
//*****
//*
//ACBCATT EXEC PGM=DFS3UACB,REGION=0M
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//SYSABEND DD SYSOUT=*
//IMS DD DSN=IMS.PSBLIB,DISP=SHR
// DD DSN=IMS.DBDLIB,DISP=SHR
//ACBCATWK DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//*
//*****
//* ACBGEN DATASETS
//*****
//IMSACB DD DSN=IMS.ACBLIB,DISP=OLD
//SYSUT3 DD UNIT=SYSDA,SPACE=(80,(100,100))
//SYSUT4 DD UNIT=SYSDA,SPACE=(256,(100,100)),DCB=KEYLEN=30
//*****
//* ACBGEN INPUT PARMS TO UPDATE ACBLIB
//*****
//SYSIN DD *
BUILD PSB=psbname
/*
//*****
//* POPULATE UTILITY DATASETS
//*****
//IMSACB01 DD DSN=*.IMSACB,DISP=OLD DO NOT REPLACE ASTERISK 22

//DFSVSAMP DD ..... BUFFER POOL DEFINITIONS
//IEFRDER DD ..... LOG DATASET FOR CATALOG UPDATES
//IEFRDER2 DD ..... LOG DATASET FOR CATALOG UPDATES
//*****
//* UPDATE INPUT PARMS FOR IMS CATALOG POPULATE UTILITY
//*****
//DFS3PPRM DD *
BMP,DFS3PU00,DFSCP001,,,,,,,,,imsid,,,,,
/*
//

```

Below is sample JCL for running the DFS3UACB (ACB Generation and Catalog Populate utility) as a DLIBATCH job. When running DFS3UACB as a DLIBATCH job, make sure the IMS catalog has either been taken off-line from IMS, or the IMS system and this batch job are part of the same shared sysplex. In this example, for more information about this utility, see [ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Utilities\)](#).

```

//ACBPOPUP JOB 'IMS SYSTEM',CLASS=K,MSGLEVEL=(1,1),REGION=0M
//*
//*****
//* DFS3UACB GENERATES ACB MEMBERS IN AN ACB LIBRARY BY CALLING THE
//* ACB MAINTENANCE UTILITY. IN THE SAME JOB STEP,
//* DFS3UACB INSERTS RECORDS IN THE EXISTING IMS CATALOG BY CALLING
//* THE IMS CATALOG POPULATE UTILITY (DFS3PU00)
//*****
//*
//ACBCATT EXEC PGM=DFS3UACB,REGION=0M
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//SYSABEND DD SYSOUT=*
//IMS DD DSN=IMS.PSBLIB,DISP=SHR
// DD DSN=IMS.DBDLIB,DISP=SHR
//ACBCATWK DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//*
//*****
//* ACBGEN DATASETS
//*****
//IMSACB DD DSN=IMS.ACBLIB,DISP=OLD
//SYSUT3 DD UNIT=SYSDA,SPACE=(80,(100,100))
//SYSUT4 DD UNIT=SYSDA,SPACE=(256,(100,100)),DCB=KEYLEN=30
//*****
//* ACBGEN INPUT PARMS TO UPDATE ACBLIB
//*****
//SYSIN DD *
BUILD PSB=psbname
//*
//*****
//* POPULATE UTILITY DATASETS
//*****
//IMSACB01 DD DSN=*.IMSACB,DISP=OLD DO NOT REPLACE ASTERISK
//DFSVSAMP DD ..... BUFFER POOL DEFINITIONS
//IEFRDER DD ..... LOG DATASET FOR CATALOG UPDATES
//IEFRDER2 DD ..... LOG DATASET FOR CATALOG UPDATES
//*****
//* UPDATE INPUT PARMS FOR IMS CATALOG POPULATE UTILITY
//*****
//DFS3PPRM DD *
DLI,DFS3PU00,DFSCP001,,,,,,,,,Y,N,,,,,,,,,DFSDF=CAT
//*
//

```

IEBCOPY

The DBDs and PSBs used for the IMS catalog are provided in the IMS RESLIB. The catalog DBDs are called DFSCD000 and DFSCX000. The PSBs are called DFSCPL00, DFSCP000, DFSCP001, DFSCP002, and DFSCP003. Below is a sample JCL that copies the DBDs and PSBs from the IMS.SDFSRESL data set to your DBDLIB and PSBLIB data sets.

```

//CPYCMEM EXEC PGM=IEBCOPY
//SYSPRINT DD SYSOUT=*
//SDFSRESL DD DSN=IMSCFG.IMB1.SDFSRESL,DISP=SHR
//DBDLIB DD DSN=IMSCFG.IMSB.DBDLIB,DISP=OLD
//PSBLIB DD DSN=IMSCFG.IMSB.PSBLIB,DISP=OLD
//SYSIN DD *
COPY OUTDD=DBDLIB,INDD=((SDFSRESL,R)),LIST=YES
SELECT MEMBER=(DFSCD000,DFSCX000)
COPY OUTDD=PSBLIB,INDD=((SDFSRESL,R)),LIST=YES
SELECT MEMBER=(DFSCPL00,DFSCP000,DFSCP001,DFSCP002,DFSCP003)
//*

```

CATALOG ACBGEN

Below is a sample JCL to perform the ACBGEN for the Catalog related PSBs. The ACBGEN is performed in the staging Library. For more information about this utility, see [Populating the IMS catalog using the ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Definition\)](#).

```
//PROCS JCLLIB ORDER=(IMSCFG.IMSB.PROCLIB)
//ACBGEN EXEC PROC=ACBGEN,SOUT='*',COMP='POSTCOMP'
//G.SYSIN DD *
BUILD PSB=(DFSCPL00)
BUILD PSB=(DFSCP000)
BUILD PSB=(DFSCP001)
BUILD PSB=(DFSCP002)
BUILD PSB=(DFSCP003)
/*
```

DSPURX00 input

The DBRC utility (DSPURX00) is used to define the Catalog database structure to the RECONS. Below are sample input commands for the DBRC DSPURX00 utility for defining an IMS Catalog with one partition. For more information about this utility, see [Database Recovery Control utility \(DSPURX00\) \(System Utilities\)](#).

```
//D.SYSIN DD *
INIT.DB DBD(DFSCD000) TYPHALDB SHARELVL(3)
INIT.PART DBD(DFSCD000) PART(DFSCD01) -
          DSNPREFIX(IMSCFG.IMSC.DFSCD000) -
          BLOCKSIZE(8192,8192,8192,8192) -
          KEYSTRNG(X'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') -
          ICJCL(ICJCL) OICJCL(OICJCL) RECOVJCL(RECOVJCL) -
          NOREUSE RECOVPD(0) GENMAX(3) RECVJCL(RECVJCL)
INIT.DB DBD(DFSCX000) TYPHALDB SHARELVL(3)
INIT.PART DBD(DFSCX000) PART(DFSCX01) -
          DSNPREFIX(IMSCFG.IMSC.DFSCX000) -
          KEYSTRNG(X'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') -
          ICJCL(ICJCL) OICJCL(OICJCL) RECOVJCL(RECOVJCL) -
          NOREUSE RECOVPD(0) GENMAX(3) RECVJCL(RECVJCL)
```

Sample DFSDFxxx

Below is a sample of the catalog section of DFSDFxxx PROCLIB member. The retention specifies 5 instances (including the active instance) and up to a year for all DBDs and PSBs. For more information about this utility, see [DFSDFxxx member of the IMS PROCLIB data set \(System Definition\)](#).

```
//*****/
/* IMS Catalog Section */
//*****/
<SECTION=CATALOG>
CATALOG=Y /*Enable IMS catalog*/
ALIAS=DFSC /*Use standard catalog prefix DFSC*/
RETENTION=(INSTANCES=5,DAYS=365) /*Retention criteria*/
//*****/
/* */
//*****/
```

DFS3PU00

Below is sample JCL for running the Catalog Populate Utility in DLIBATCH mode. For more information about this utility, see [Populating the IMS catalog using the IMS Catalog Populate utility \(DFS3PU00\) \(System Definition\)](#).

```
//UPDTCAT EXEC PGM=DFS3PU00,
// PARM=(DLI,DFS3PU00,DFSCPL00,,,,,,,,Y,N,,,,,,,,,'DFSDF=001')
//STEPLIB DD DSN= IMSCFG.IMB1.SDFSRESL,DISP=SHR
//DFSRESLB DD DSN=IMS.PROCLIB,DISP=SHR
//IMS DD DSN= IMSCFG.IMSB.PSBLIB,DISP=SHR
// DD DSN= IMSCFG.IMSB.DBDLIB,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSABEND DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//IEFRDER DD DISP=(,CATLG),DSN=IMSCFG.IMSB.PU000.LOG(+1),
// SPACE=(TRK,(5,5),RLSE),UNIT=SYSALLDA
//DFSVSAMP DD DSN= IMSCFG.IMSB.PROCLIB(DFSVS MDB),DISP=SHR
//IMSACB01 DD DSN= IMSCFG.IMSB.ACBLIBA,DISP=SHR
//IMSACB02 DD DSN= IMSCFG.IMSB.ACBLIB,DISP=SHR
```

Sample database structure

This sample database structure depicts the IMS database layout for the sample Insurance company for open access.

The CUSTOMER table contains information about the insurance company customer. Each customer can have many policies which are stored in the POLICY table. If there are any claims associated with a policy, that information would be stored in the CLAIMS table. In addition a customer can have zero to many dependents such as a child or spouse which would be stored in the DEPENDENT table. All communications with a customer whether through a phone operator, an online chat rep, or the email system will be stored in the COMMUNICATION table.

DBD name: INSURDB | Database access method: (HIDAM,VSAM)

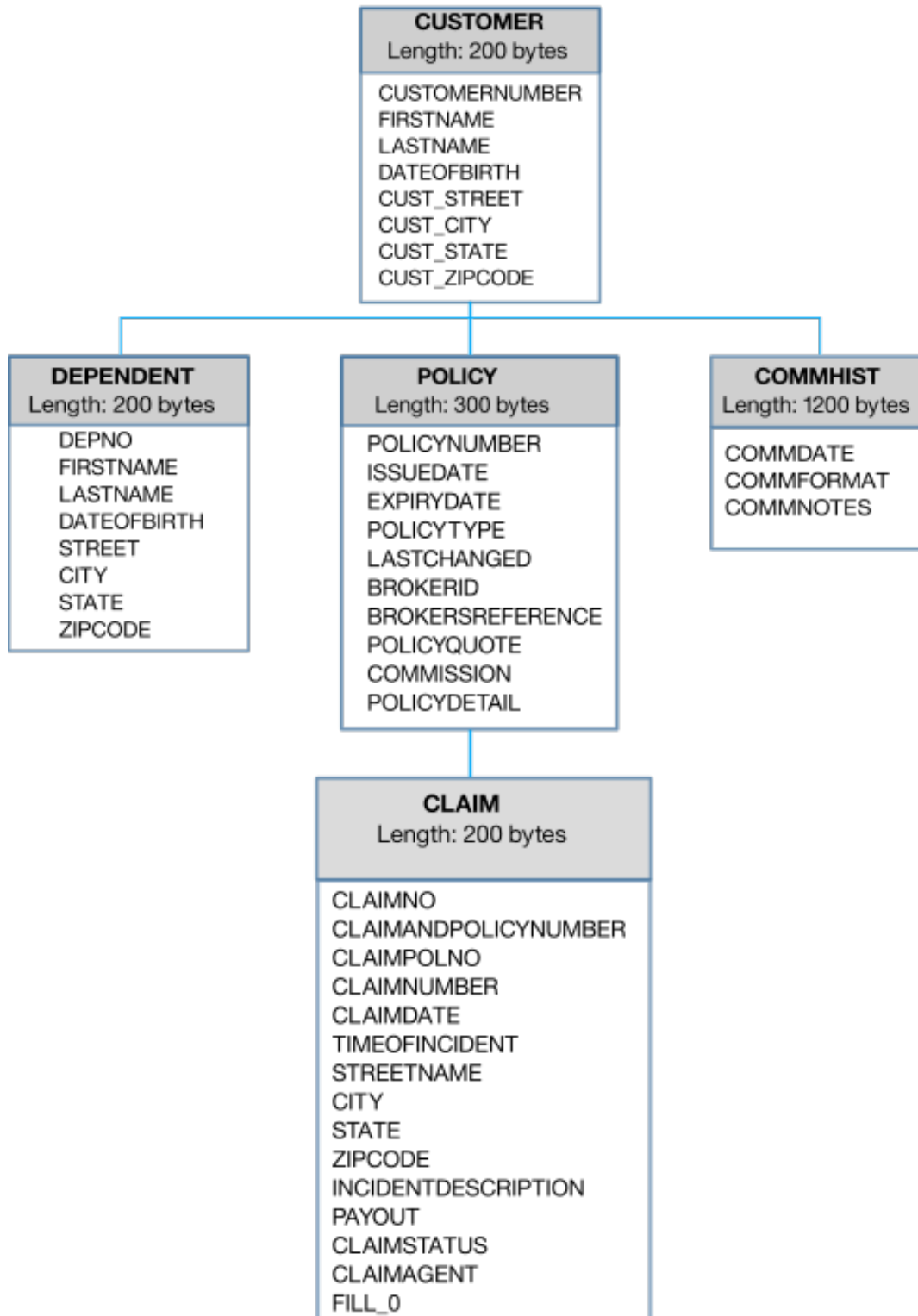


Figure 2. Database structure for the sample IMS Insurance company

Accessing IMS data using Java applications

If you have Java applications outside of IMS that need access to the data, whether it is for querying or analytics or any purpose, the open access kit can guide you to make that happen.

For more information, see [IMS Open Access](#).

Use case: Using Java applications

The sample application used in this use case defines a web service (using JAX-RS 2.0) to retrieve customer information about insurance policies for the given customer number. The application makes a connection to the IMS insurance database using the @Resource annotation to provide the connection factory into the application as a JDBC DataSource. This DataSource can then be used to access IMS data using SQL calls.

Goal:

Prosentient Insurance, a fictitious company, intends to develop a new Claim Handling client portal, which requires that IMS data be easily accessible (open) to the Java apps that will comprise the new portal.

Preconditions:

- Data is located across multiple platforms
- A lack of integration between key systems has contributed to the current sluggish client experience
- Prosentient has more Java skills in house than COBOL skills

Success End Condition:

Java Developers can easily create and deploy apps that access IMS data for the new portal, without requiring deep knowledge of the hierarchical data model or IMS's DL/I access language.

Primary Actors:

- Vikram, System Programmer at Prosentient
- Susan, Application Developer at Prosentient
- Steve, Database Administrator at Prosentient

Secondary Actors:

- Ana, Enterprise Architect at Prosentient
- Lars, an independent industry consultant to Prosentient
- Matt, Project Manager at Prosentient
- Trish, SAF Administrator at Prosentient

Main Use Case Success Scenario:

1. Each of the people involved in the project familiarize themselves with the overall structure of the open access architecture, with particular focus on the areas for which they have responsibility.
2. Vikram sets up the Common Service Layer infrastructure required for open access to IMS data.
3. Vikram sets up the IMS Catalog
4. Steve downloads and installs the IMS Explorer
5. Steve captures the metadata for IMS databases using IMS Explorer
6. Susan verifies the IMS database schema using IMS Explorer. As an application developer, she would most likely be familiar with all of the application databases.
7. Susan creates and executes SQL queries using IMS Explorer

8. Susan deploys IMS database resource adapters in WebSphere Liberty Server
9. Susan develops a Java application that uses the SQL queries to access IMS data

Related information

[Getting started with open access](#)

Implementing open access for Java applications

The first step in getting started with open access to IMS data is to set up the open access infrastructure, which provides access to Java™ applications. You also need to download and install IMS Explorer for Development, capture the database metadata, and connect to the IMS database. Once these are in place, you can develop applications that use standard JDBC programming calls to execute SQL queries.

About this task

It is important to understand that although SQL syntax is used by the application, the IMS Universal JDBC driver translates these SQL statements to DL/I calls and formats the data returned into SQL result sets for the application.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

Table 2. Implementing open access	
Procedure	User role who completes the task
1. Learn and get started with open access. 1. “Use case: Using Java applications” on page 20 (topic) 2. Getting Started with opening access to IMS data (video)	Everyone on the team, to get an understanding of what is involved
2. Set up the infrastructure needed for open access to IMS database. 1. Setting up open access infrastructure (article) <ul style="list-style-type: none"> • “Sample code for getting started with open access” on page 10 (topic, sample code) 2. Setting up the IMS catalog (article) <ul style="list-style-type: none"> • “Sample code for setting up the IMS Catalog” on page 29 (topic, sample code) 3. “Sample Java architectural diagram” on page 25 (topic, sample code)	Vikram, the System Programmer
3. Install IMS Explorer for Development (topic)	Steve, the DBA and Susan, the Application Developer
4. Capturing Metadata for IMS data (video)	Steve, the DBA
5. Capturing IMS Metadata: Advanced (video)	Steve, the DBA
6. Explore, Visualize and Understand IMS database schema (video) <ul style="list-style-type: none"> • “Sample database structure” on page 33 (topic, sample code) 	Susan, the Application Developer
7. Creating and executing SQL Queries with IMS Explorer (video)	Susan, the Application Developer

Table 2. Implementing open access (continued)	
Procedure	User role who completes the task
8. Deploying IMS Database Resource Adapters (video) <ul style="list-style-type: none"> • “Sample Java application deployment for WebSphere Liberty” on page 22 (topic, sample code) 	Susan, the Application Developer
9. Develop applications to access IMS data (video) <ul style="list-style-type: none"> • “Sample Java application code for our use case” on page 23 (topic, sample code) 	Susan, the Application Developer

Samples overview for Java

The following sample code pertains to the use case scenario for the insurance company. Use these Java samples to understand how to implement open access and how to develop the application that queries the database.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

Sample Java application deployment for WebSphere Liberty

This sample server.xml configuration file is for WebSphere Liberty, to deploy the IMS Universal Database Resource Adapters, define a connection factory to connect to an IMS insurance database, and deploy the RESTful service application.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

Once deployed you can invoke the REST service using a URL similar to the following:

`http://<host>:<port>/Insurance/services/customer/info?customerNumber=3`

In the following sample, the text in **bold** shows:

- The **resourceAdapter** tag defines the IMS Universal Database Resource Adapter to WebSphere Liberty.
- The **connectionFactory** tag defines the connection factory to WebSphere Liberty.
- The **webApplication** tag defines the REST service for retrieving customer information about insurance policies for the given customer number.


```

<server description="new server">

    <!-- Enable features -->
    <featureManager>
        <feature>localConnector-1.0</feature>
        <feature>jaxrs-2.0</feature>
        <feature>ejbLite-3.2</feature>
        <feature>jca-1.7</feature>
        <feature>jndi-1.0</feature>
        <feature>jdbc-4.1</feature>
        <feature>ssl-1.0</feature>
    </featureManager>

    <!-- This template enables security. To get the full use of all the capabilities, a keystore
    and user registry are required. -->

    <!-- For the keystore, default keys are generated and stored in a keystore. To provide the
    keystore password, generate an
    encoded password using bin/securityUtility encode and add it below in the password
    attribute of the keyStore element.
    Then uncomment the keyStore element. -->
    <!--
    <keyStore password="" />
    -->

    <!--For a user registry configuration, configure your user registry. For example, configure a
    basic user registry using the
    basicRegistry element. Specify your own user name below in the name attribute of the user
    element. For the password,
    generate an encoded password using bin/securityUtility encode and add it in the password
    attribute of the user element.
    Then uncomment the user element. -->
    <basicRegistry id="basic" realm="BasicRealm">
        <!-- <user name="yourUserName" password="" /> -->
    </basicRegistry>

    <!-- To access this server from a remote client add a host attribute to the following
    element, e.g. host="" -->
    <httpEndpoint httpPort="9080" httpsPort="9443" id="defaultHttpEndpoint"/>

    <!-- Automatically expand WAR files and EAR files -->
    <applicationManager autoExpand="true"/>

    <resourceAdapter id="imsudbJLocal" location="/Users/user/Desktop/IMS/imsudbJLocal.rar"/>
    <connectionFactory jndiName="INSURANCEDB">
        <properties.imsudbJLocal databaseName="INSUR01" dataStoreName="IMDA"
    dataStoreServer="ims.system.hostname.com" driverType="4" password="APASSWD" portNumber="6690"
    user="AUSER"/>
    </connectionFactory>
    <keyStore password="{xor}PDc+MTg6Nis="/>

    <applicationMonitor updateTrigger="mbean"/>

    <webApplication id="Insurance" location="Insurance.war" name="Insurance"/>
</server>

```

Sample Java application code for our use case

This sample application utilizes JAX-RS 2.0 to define a web service to retrieve customer information about insurance policies for the given customer number provided. The application gets a connection to the IMS insurance database using the @Resource annotation to inject the connection factory into the application as a JDBC DataSource. This DataSource is then used to access IMS data using SQL calls.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

```

package customer.info;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;

import javax.annotation.Resource;
import javax.ejb.Stateless;
import javax.sql.DataSource;
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.Path;

```

```

import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;

import policy.info.AutoPolicy;
import policy.info.HousePolicy;

@Stateless
@Path("/customer")
public class CustomerInfoService {
    @Resource(name="INSURANCEDB")
    DataSource insuranceDB;

    @Path("/info")
    @GET
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces({ MediaType.APPLICATION_JSON })
    public CustomerInfo getCustomerInfo(@QueryParam("customerNumber") int customerNumber) {
        CustomerInfo customerInfo = new CustomerInfo();
        try {
            Connection conn = insuranceDB.getConnection();
            PreparedStatement customerStatement = conn.prepareStatement("SELECT FIRSTNAME, LASTNAME,
CUST_STREET, CUST_CITY, CUST_STATE, CUST_ZIPCODE, DATEOFBIRTH FROM INSURPCB.CUSTOMER WHERE
CUSTOMERNUMBER = ?");
            PreparedStatement autoPolicyStatement = conn.prepareStatement("SELECT POLICYNUMBER,
CAR_MAKE, CAR_MODEL, CAR_MANUFACTUREDATE, CAR_REGNUMBER, CAR_DRIVENAME FROM INSURPCB.POLICY
WHERE CUSTOMER_CUSTNO = ? and POLICYTYPE='A'");
            PreparedStatement housePolicyStatement = conn.prepareStatement("SELECT POLICYNUMBER,
HOMEPROPERTYTYPE, HOMEBEDROOMS, HOMEHOUSEVALUE, HOME_STREET, HOME_ZIPCODE FROM INSURPCB.POLICY
WHERE CUSTOMER_CUSTNO = ? and POLICYTYPE='H'");

            customerStatement.setInt(1, customerNumber);
            autoPolicyStatement.setInt(1, customerNumber);
            housePolicyStatement.setInt(1, customerNumber);

            ResultSet customerRS = customerStatement.executeQuery();
            if(customerRS.next()){
                customerInfo.setFirstName(customerRS.getString(1).trim());
                customerInfo.setLastName(customerRS.getString(2).trim());
                customerInfo.setStreet(customerRS.getString(3).trim());
                customerInfo.setCity(customerRS.getString(4).trim());
                customerInfo.setState(customerRS.getString(5).trim());
                customerInfo.setZipCode(customerRS.getString(6).trim());
                customerInfo.setDateOfBirth(customerRS.getString(7).trim());
                customerRS.close();
                customerStatement.close();

                //Collect House Policy Information
                ResultSet houseRS = housePolicyStatement.executeQuery();
                while(houseRS.next()){
                    HousePolicy housePolicy = new HousePolicy();
                    housePolicy.setPolicyNumber(houseRS.getInt(1));
                    housePolicy.setPropertyType(houseRS.getString(2).trim());
                    housePolicy.setNumberOfBedrooms(houseRS.getShort(3));
                    housePolicy.setValue(houseRS.getInt(4));
                    housePolicy.setStreet(houseRS.getString(5).trim());
                    housePolicy.setZipCode(houseRS.getString(6).trim());
                    customerInfo.addHousePolicy(housePolicy);
                }
                houseRS.close();
                housePolicyStatement.close();

                //Collect Auto Policy Information
                ResultSet autoRS = autoPolicyStatement.executeQuery();
                while(autoRS.next()){
                    AutoPolicy autoPolicy = new AutoPolicy();
                    autoPolicy.setPolicyNumber(autoRS.getInt(1));
                    autoPolicy.setMake(autoRS.getString(2).trim());
                    autoPolicy.setModel(autoRS.getString(3).trim());
                    autoPolicy.setManufactureDate(autoRS.getString(4).trim());
                    autoPolicy.setRegistrationNumber(autoRS.getString(5).trim());
                    autoPolicy.setDriverName(autoRS.getString(6).trim());
                    customerInfo.addAutoPolicy(autoPolicy);
                }
                autoRS.close();
                autoPolicyStatement.close();
            }
        } catch (Exception e) {
            e.printStackTrace();
        }

        return customerInfo;
    }
}

```

}
}

Sample Java architectural diagram

This sample architectural diagram shows how Ana's team is going to implement the open access solution.

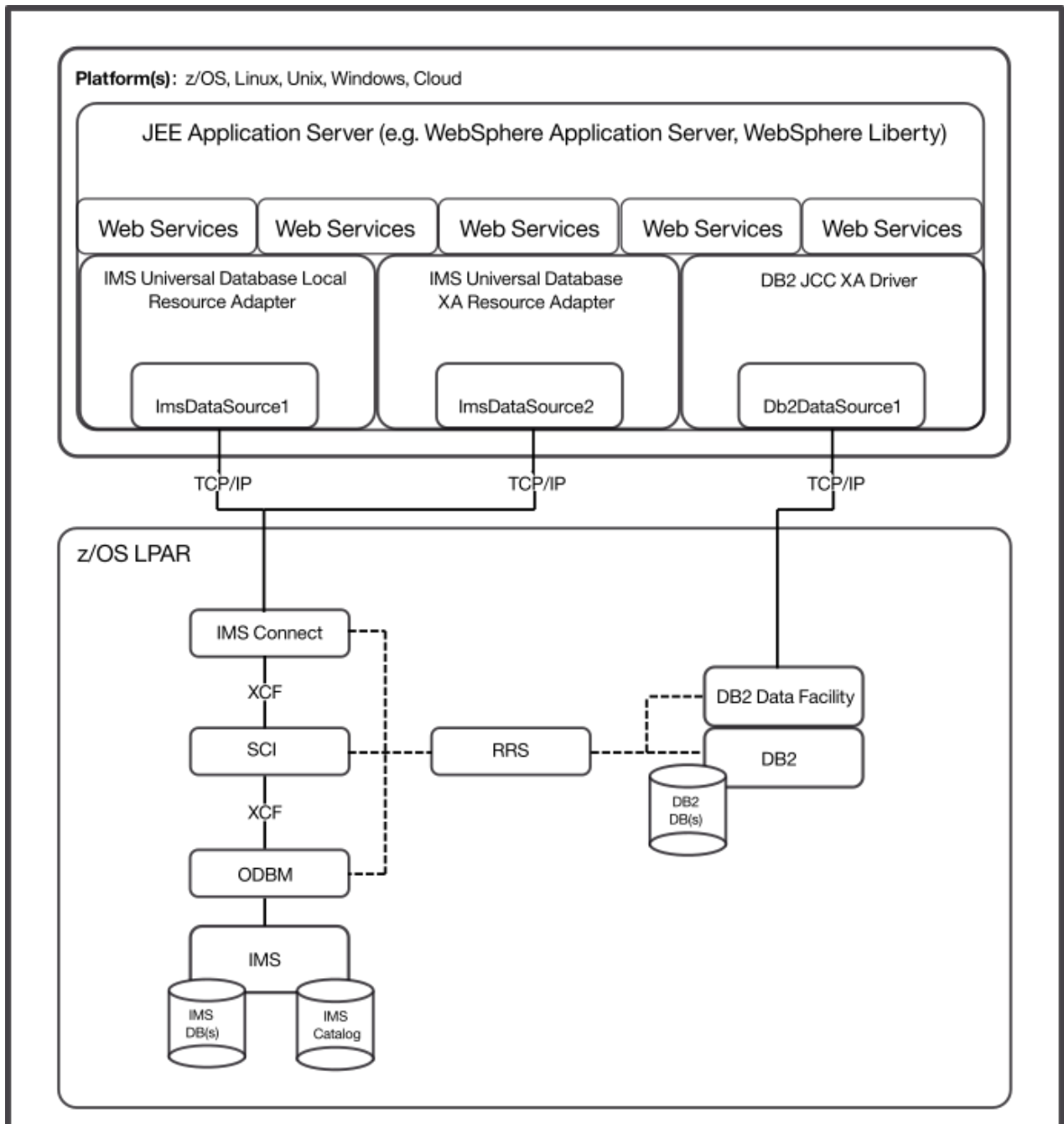


Figure 3. High level architectural diagram of open access.

Related information

Meet Lars and the Prosentient Team and see how they intend to open access to IMS data

Sample code for getting started with open access

This code shows sample configurations for procedures, configuration members and JCL for setting up an IMS system to provide open access.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

IMS Connect JCL and PROCLIB member

IMS Connect JCL and PROCLIB member contain the following sample code:

HWSCFG00 PROCLIB member: Below are sample IMS Connect Configuration parameters. The ODACCESS statement specifies parameters related to IMS open access. Here, the port number is 5555, port timeout value is set to 60 seconds, ODBM timeout is also set to 60 secs. The IMSplex name is Plex1 and the IMS Connect name within that IMSplex is IVP14HWS. The **ODBAUTOCONN=Y** parameter specifies that IMS Connect will automatically connect to new and existing instances of ODBM within that IMSplex.

```
HWS=(ID=IVP14HWS,XIBAREA=100,RACF=N,RRS=Y)
TCPIP=(HOSTNAME=TCPIP,PORTID=(9999,LOCAL),RACFID=RACFUID1,TIMEOUT=5000)
DATASTORE=(GROUP=IVPXCFCGN,ID=IVP1,MEMBER=IVP14HWS,DRU=HWSYDRU0,
TMEMBER=IVP1)
ODACCESS=(ODBAUTOCONN=Y,IMSPLEX=(MEMBER=IVP14HWS,TMEMBER=PLEX1),
DRDAPORT=(ID=5555,PORTTMO=6000),ODBMTMO=6000)
```

IMS Connect Startup PROCLIB: Below is sample JCL to start the IMS Connect address space.

Note: We are using **HWSCFG00** for the IMS Connect configuration parameters and **BPECFGHT** for the BPE configuration parameters.

```
//HWS1      PROC  RGN=0M,SOUT=A,
//          BPECFG=BPECFGHT,
//          HWSCFG=HWSCFG00
// *
// *****
// * BRING UP AN IMS CONNECT USING BPEINI00
// *****
//STEP1     EXEC  PGM=BPEINI00,REGION=0M,TIME=1440,
//          PARM='BPECFG=&BPECFG,BPEINIT=HWSINI00,
//          HWSCFG=&HWSCFG'
//STEPLIB  DD   DSN= IMS.SDFSRESL,DISP=SHR
//PROCLIB  DD   DSN= IMS.PROCLIB,DISP=SHR
//SYSPRINT DD   SYSOUT=&SOUT
//SYSUDUMP DD   SYSOUT=&SOUT
```

IMS PROCLIB member for IMSplex

IMS PROCLIB member for IMSplex contains the following sample code:

DFSCGxxx PROCLIB member: The DFSCGxxx PROCLIB member contains various Common Service Layer (CSL) parameters. In this IMS open access solution adoption kit, there are two parameters needed:

1. **IMSPLEX=plex1** specifies the name of our IMSplex as plex1.
2. **RMENV=N** specifies that the Resource Manager (RM) component of CSL is not to be used.

The suffix for DFSCGxxx, xxx, is specified on the **CSLG=** parameter in the IMSPBxxx PROCLIB member.

```
IMSPLEX=plex1,
RMENV=N
```

ODBM JCL and PROCLIB members

ODBM JCL and PROCLIB members contain the following sample code:

BPEODBM PROCLIB member: Below is sample BPE configuration PROCLIB member for ODBM. In this member, the internal dispatcher, AWE, CSL and ODBM traces are set to HIGH and all other internal traces are set to Low. These traces are typically examined by the IMS support team for problem analysis.

```
# BPE CONFIGURATION MEMBER for ODBM - BPEODBM
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE) /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE */
TRCLEV=(AWE,HIGH,BPE) /* AWE SERVER TRACE */
# ODBM Traces
TRCLEV=(*,LOW,ODBM)
TRCLEV=(CSL,HIGH,ODBM)
TRCLEV=(ODBM,HIGH,ODBM)
```

CSLDC001 PROCLIB member: Below are sample parameters for the CSLDCxxx PROCLIB member. In the **LOCAL_DATASTORE_CONFIGURATION** section, ODBM (with the name of OD1) will connect to an IMS datastore with an ID of IMS1. The IMS datastore is IMS1, and the ALIASes used by the applications are: IM1A and IM1B.

```
*****
* CSLDC001 ODBM CSL PROCLIB MEMBER *
*****
<SECTION=GLOBAL_DATASTORE_CONFIGURATION>

IDRETRY=5 /* Retry connection 5 times before quit */
MAXTHRDS=50 /* 10 threads max to any IMS Datastore */
TIMER=30 /* 30 seconds between ID retry attempts */
FPBUF=10 /* 10 DEDB buffers per thread */
FPBOF=10 /* 10 Overflow buffers per thread */
CNBA=200 /* (FPBUF+FPBOF)*MAXTHRDS <= CNBA */
*****
* Define DATASTORE properties for ODBM name OD1 *
*****
<SECTION=LOCAL_DATASTORE_CONFIGURATION>
ODBM(NAME=OD1, /* Define parms for OD1 */
  DATASTORE(NAME=IMS1, /* IMSID on LPAR A */
    ALIAS(NAME=IM1A,NAME=IM1B), /* Names for APPL sets 1 & 2 */
  )
)
```

CSLDI001 PROCLIB member: Below is sample CSLDIxxx PROCLIB member. In this member, the **IMSPLEX(NAME=PLEX1)** parameter specifies the IMSplex name (PLEX1) that ODBM participates in. The **ODBMCFG=001** parameter specifies the suffix for the CSLDCxxx PROCLIB member. The **ODBMNAME=OD1** parameter specifies OD1 as the name of this ODBM.

```
*-----*
* ODBM INITIALIZATION PROCLIB MEMBER - CSLDI001 *
*-----*
ARMRST=N, /* SHOULD ARM RESTART OM ON FAILURE */
IMSPLEX(NAME=PLEX1) /* IMSPLEX NAME (CSLPLEX1) */
ODBMCFG=001 /* suffix for CSLDCxxx (CSLDC001) */
ODBMNAME=OD1, /* ODBM NAME (ODBMID = OD10D) */
RRS=Y /* RRS=Y will use ODBA access */
*-----*
```

ODBM Startup PROC: Below is sample JCL to startup the ODBM address space.

Note: Below is a sample JCL procedure to start the ODBM address space. The PROCLIB member BPEODBM is being used for the BPE configuration parameters, and the ODBMINIT=001 parameter indicates the PROCLIB member CSLDI001 is being used for the ODBM initialization parameters.

```

/*-----*/
/* ODBM Startup Proc */
/*-----*/
//IMS10DBM PROC RGN=3000K,SOUT=X,
//                      BPECFG=BPE0DBM,
//                      ODBMINIT=001
//
/*
//IMS10DBM EXEC PGM=BPEINI00,REGION=&RGN,
// PARM=('BPECFG=&BPECFG','BPEINIT=CSLDINI0','ODBMINIT=&ODBMINIT')
//
/*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//
/*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT

```

OM JCL and PROCLIB members

OM JCL and PROCLIB members contain the following sample code:

BPECONFG PROCLIB member: Below is a sample BPE configuration PROCLIB member that is used by the OM address space.

```

# BPE CONFIG MEMBER for SCI & OM - BPECONFG
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE) /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE */
TRCLEV=(AWE,HIGH,BPE) /* AWE SERVER TRACE */

```

CSLOI000 PROCLIB member: Below is a sample CSLOIxxx PROCLIB member that contains the OM initialization parameters.

```

*-----*
* OM INITIALIZATION PROCLIB MEMBER - CSLOI000 *
*-----*
ARMRST=N, /* SHOULD ARM RESTART OM ON FAILURE */
CMDLANG=ENU, /* USE ENGLISH FOR COMMAND DESC */
CMDSEC=N, /* NO COMMAND SECURITY */
OMNAME=OM1, /* OM NAME (OMID = OM10M) */
IMSPLEX(
  NAME=PLEX1, /* IMSPLEX NAME (CSLPLEX1) */
  AUDITLOG=SYSLOG.OM2Q01.LOG), /* MVS LOG STREAM */
CMDTEXTDSN=SSTBLD.I14APAR.DBDC.M.CM1.SDFSDATA /* CMD TXT DATAS */
*-----*

```

OM Startup PROC: Below is a sample JCL procedure to start the OM address space. It uses the BPECONFG PROCLIB member for the BPE configuration parameters and **OMINIT=000** specifies the suffix for the CSLOIxxx PROCLIB member.

```

/*-----*/
/* OM Startup Proc */
/*-----*/
//IMS10M PROC RGN=3000K,SOUT=X,
//                      BPECFG=BPECONFG,
//                      OMINIT=000
//
/*
//IMS10M EXEC PGM=BPEINI00,REGION=&RGN,
// PARM=('BPECFG=&BPECFG','BPEINIT=CSLOINI0','OMINIT=&OMINIT')
//
/*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//
/*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT

```

SCI JCL and PROCLIB members

SCI JCL and PROCLIB members provide the following sample code:

BPECONFG PROCLIB member: Below is a sample BPE configuration PROCLIB member that is used by the SCI address space.

Note: This is the same member that is used by the OM address space.

```
# BPE CONFIG MEMBER for SCI & OM - BPECONFG
LANG=ENU
#
# DEFINITIONS FOR BPE SYSTEM TRACES
#
TRCLEV=(*,LOW,BPE)                /* DEFAULT ALL TRACES TO LOW */
TRCLEV=(DISP,HIGH,BPE,PAGES=12)) /* DISPATCHER TRACE */
TRCLEV=(AWE,HIGH,BPE)             /* AWE SERVER TRACE */
```

CSLSI000 PROCLIB member: Below is sample CSLSIxxx PROCLIB member that contains the SCI initialization parameters. The **IMSPLEX(NAME=PLEX1)** parameter specifies the IMSplex name that SCI participates in. The **SCINAME=SCI1** parameter specifies the SCI address space name.

```
*-----*
* SCI INITIALIZATION PROCLIB MEMBER - CSLSI000 *
*-----*
ARMRST=N,                /* SHOULD ARM RESTART SCI ON FAILURE */
IMSPLEX(NAME=PLEX1)       /* IMSPLEX NAME (CSLPLEX1) */
SCINAME=SCI1,            /* SCI NAME (SCIID = SCI1SCI) */
```

SCI Startup PROC: Below is a sample JCL procedure to start the SCI address space. It uses the BPECONFG PROCLIB member for the BPE configuration parameters, and **SCIINIT=000** specifies the suffix for the CSLSIxxx PROCLIB member, which in this case is pointing to CSLSI000.

```
//*-----*
/* SCI Startup Proc */
/*-----*
//IMS1SCI PROC RGN=3000K,SOUT=X,
//                BPECFG=BPECONFG,
//                SCIINIT=000
//*
//IMS1SCI EXEC PGM=BPEINI00,REGION=&RGN,
// PARM=('BPECFG=&BPECFG','BPEINIT=CSLSINI0','SCIINIT=&SCIINIT')
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//*
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSPRINT DD SYSOUT=&SOUT
```

Sample code for setting up the IMS Catalog

These sample configurations for procedures, configuration members and JCL show how to set up the IMS catalog.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

DFS3UACB

Below is a sample JCL for running the DFS3UACB (ACB Generation and Catalog Populate utility) as a BMP job. You can run DFS3UACB as a BMP when the IMS catalog is being used by the IMS online system. Ensure that your IMS system has the catalog database open for update; the default is read-only. For more information about this utility, see [ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Utilities\)](#).

```

//ACBPOPUP JOB 'IMS SYSTEM',CLASS=K,MSGLEVEL=(1,1),REGION=0M
//*
//*****
//* DFS3UACB GENERATES ACB MEMBERS IN AN ACB LIBRARY BY CALLING THE
//* ACB MAINTENANCE UTILITY. IN THE SAME JOB STEP,
//* DFS3UACB INSERTS RECORDS IN THE EXISTING IMS CATALOG BY CALLING
//* THE IMS CATALOG POPULATE UTILITY (DFS3PU00)
//*****
//*
//ACBCATT EXEC PGM=DFS3UACB,REGION=0M
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//SYSABEND DD SYSOUT=*
//IMS DD DSN=IMS.PSBLIB,DISP=SHR
// DD DSN=IMS.DBDLIB,DISP=SHR
//ACBCATWK DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//*
//*****
//* ACBGEN DATASETS
//*****
//IMSACB DD DSN=IMS.ACBLIB,DISP=OLD
//SYSUT3 DD UNIT=SYSDA,SPACE=(80,(100,100))
//SYSUT4 DD UNIT=SYSDA,SPACE=(256,(100,100)),DCB=KEYLEN=30
//*****
//* ACBGEN INPUT PARMS TO UPDATE ACBLIB
//*****
//SYSIN DD *
BUILD PSB=psbname
//*
//*****
//* POPULATE UTILITY DATASETS
//*****
//IMSACB01 DD DSN=*.IMSACB,DISP=OLD DO NOT REPLACE ASTERISK 22

//DFSVSAMP DD ..... BUFFER POOL DEFINITIONS
//IEFRDER DD ..... LOG DATASET FOR CATALOG UPDATES
//IEFRDER2 DD ..... LOG DATASET FOR CATALOG UPDATES
//*****
//* UPDATE INPUT PARMS FOR IMS CATALOG POPULATE UTILITY
//*****
//DFS3PPRM DD *
BMP,DFS3PU00,DFSCP001,,,,,,,,,imsid,,,,,
//
//

```

Below is sample JCL for running the DFS3UACB (ACB Generation and Catalog Populate utility) as a DLIBATCH job. When running DFS3UACB as a DLIBATCH job, make sure the IMS catalog has either been taken off-line from IMS, or the IMS system and this batch job are part of the same shared sysplex. In this example, for more information about this utility, see [ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Utilities\)](#).


```

//ACBPOPUP JOB 'IMS SYSTEM',CLASS=K,MSGLEVEL=(1,1),REGION=0M
//*
//*****
//* DFS3UACB GENERATES ACB MEMBERS IN AN ACB LIBRARY BY CALLING THE
//* ACB MAINTENANCE UTILITY. IN THE SAME JOB STEP,
//* DFS3UACB INSERTS RECORDS IN THE EXISTING IMS CATALOG BY CALLING
//* THE IMS CATALOG POPULATE UTILITY (DFS3PU00)
//*****
//*
//ACBCATT EXEC PGM=DFS3UACB,REGION=0M
//*
//STEPLIB DD DSN=IMS.SDFSRESL,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//DFSRESLB DD DSN=IMS.SDFSRESL,DISP=SHR
//SYSPRINT DD SYSOUT=A
//SYSOUT DD SYSOUT=A
//SYSABEND DD SYSOUT=*
//IMS DD DSN=IMS.PSBLIB,DISP=SHR
// DD DSN=IMS.DBDLIB,DISP=SHR
//ACBCATWK DD SPACE=(CYL,(1,1)),UNIT=SYSDA
//*
//*****
//* ACBGEN DATASETS
//*****
//IMSACB DD DSN=IMS.ACBLIB,DISP=OLD
//SYSUT3 DD UNIT=SYSDA,SPACE=(80,(100,100))
//SYSUT4 DD UNIT=SYSDA,SPACE=(256,(100,100)),DCB=KEYLEN=30
//*****
//* ACBGEN INPUT PARMS TO UPDATE ACBLIB
//*****
//SYSIN DD *
BUILD PSB=psbname
/*
//*****
//* POPULATE UTILITY DATASETS
//*****
//IMSACB01 DD DSN=*.IMSACB,DISP=OLD DO NOT REPLACE ASTERISK
//DFSVSAMP DD ..... BUFFER POOL DEFINITIONS
//IEFRDER DD ..... LOG DATASET FOR CATALOG UPDATES
//IEFRDER2 DD ..... LOG DATASET FOR CATALOG UPDATES
//*****
//* UPDATE INPUT PARMS FOR IMS CATALOG POPULATE UTILITY
//*****
//DFS3PPRM DD *
DLI,DFS3PU00,DFSCP001,,,,,,,,,Y,N,,,,,,,,,DFSDF=CAT
/*
//

```

IEBCOPY

The DBDs and PSBs used for the IMS catalog are provided in the IMS RESLIB. The catalog DBDs are called DFSCD000 and DFSCX000. The PSBs are called DFSCPL00, DFSCP000, DFSCP001, DFSCP002, and DFSCP003. Below is a sample JCL that copies the DBDs and PSBs from the IMS.SDFSRESL data set to your DBDLIB and PSBLIB data sets.

```

//CPYCMEM EXEC PGM=IEBCOPY
//SYSPRINT DD SYSOUT=*
//SDFSRESL DD DSN=IMSCFG.IMB1.SDFSRESL,DISP=SHR
//DBDLIB DD DSN=IMSCFG.IMSB.DBDLIB,DISP=OLD
//PSBLIB DD DSN=IMSCFG.IMSB.PSBLIB,DISP=OLD
//SYSIN DD *
COPY OUTDD=DBDLIB,INDD=((SDFSRESL,R)),LIST=YES
SELECT MEMBER=(DFSCD000,DFSCX000)
COPY OUTDD=PSBLIB,INDD=((SDFSRESL,R)),LIST=YES
SELECT MEMBER=(DFSCPL00,DFSCP000,DFSCP001,DFSCP002,DFSCP003)
/*

```

CATALOG ACBGEN

Below is a sample JCL to perform the ACBGEN for the Catalog related PSBs. The ACBGEN is performed in the staging Library. For more information about this utility, see [Populating the IMS catalog using the ACB Generation and Catalog Populate utility \(DFS3UACB\) \(System Definition\)](#).

```
//PROCS JCLLIB ORDER=(IMSCFG.IMSB.PROCLIB)
//ACBGEN EXEC PROC=ACBGEN,SOUT='*',COMP='POSTCOMP'
//G.SYSIN DD *
BUILD PSB=(DFSCPL00)
BUILD PSB=(DFSCP000)
BUILD PSB=(DFSCP001)
BUILD PSB=(DFSCP002)
BUILD PSB=(DFSCP003)
/*
```

DSPURX00 input

The DBRC utility (DSPURX00) is used to define the Catalog database structure to the RECONS. Below are sample input commands for the DBRC DSPURX00 utility for defining an IMS Catalog with one partition. For more information about this utility, see [Database Recovery Control utility \(DSPURX00\) \(System Utilities\)](#).

```
//D.SYSIN DD *
INIT.DB DBD(DFSCD000) TYPHALDB SHARELVL(3)
INIT.PART DBD(DFSCD000) PART(DFSCD01) -
          DSNPREFIX(IMSCFG.IMSC.DFSCD000) -
          BLOCKSIZE(8192,8192,8192,8192) -
          KEYSTRNG(X'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') -
          ICJCL(ICJCL) OICJCL(OICJCL) RECOVJCL(RECOVJCL) -
          NOREUSE RECOVPD(0) GENMAX(3) RECVJCL(RECVJCL)
INIT.DB DBD(DFSCX000) TYPHALDB SHARELVL(3)
INIT.PART DBD(DFSCX000) PART(DFSCX01) -
          DSNPREFIX(IMSCFG.IMSC.DFSCX000) -
          KEYSTRNG(X'FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF') -
          ICJCL(ICJCL) OICJCL(OICJCL) RECOVJCL(RECOVJCL) -
          NOREUSE RECOVPD(0) GENMAX(3) RECVJCL(RECVJCL)
```

Sample DFSDFxxx

Below is a sample of the catalog section of DFSDFxxx PROCLIB member. The retention specifies 5 instances (including the active instance) and up to a year for all DBDs and PSBs. For more information about this utility, see [DFSDFxxx member of the IMS PROCLIB data set \(System Definition\)](#).

```
//*****/
/* IMS Catalog Section */
/*****/
<SECTION=CATALOG>
CATALOG=Y /*Enable IMS catalog*/
ALIAS=DFSC /*Use standard catalog prefix DFSC*/
RETENTION=(INSTANCES=5,DAYS=365) /*Retention criteria*/
/*****/
/* */
/*****/
```

DFS3PU00

Below is sample JCL for running the Catalog Populate Utility in DLIBATCH mode. For more information about this utility, see [Populating the IMS catalog using the IMS Catalog Populate utility \(DFS3PU00\) \(System Definition\)](#).

```
//UPDTCAT EXEC PGM=DFS3PU00,
// PARM=(DLI,DFS3PU00,DFSCPL00,,,,,,,,Y,N,,,,,,,,,'DFSDF=001')
//STEPLIB DD DSN= IMSCFG.IMB1.SDFSRESL,DISP=SHR
//DFSRESLB DD DSN=IMS.PROCLIB,DISP=SHR
//IMS DD DSN= IMSCFG.IMSB.PSBLIB,DISP=SHR
// DD DSN= IMSCFG.IMSB.DBDLIB,DISP=SHR
//PROCLIB DD DSN=IMS.PROCLIB,DISP=SHR
//SYSABEND DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//IEFRDER DD DISP=(,CATLG),DSN=IMSCFG.IMSB.PU000.LOG(+1),
// SPACE=(TRK,(5,5),RLSE),UNIT=SYSALLDA
//DFSVSAMP DD DSN= IMSCFG.IMSB.PROCLIB(DFSVSMDB),DISP=SHR
//IMSACB01 DD DSN= IMSCFG.IMSB.ACBLIBA,DISP=SHR
//IMSACB02 DD DSN= IMSCFG.IMSB.ACBLIB,DISP=SHR
```

Sample database structure

This sample database structure depicts the IMS database layout for the sample Insurance company for open access.

The CUSTOMER table contains information about the insurance company customer. Each customer can have many policies which are stored in the POLICY table. If there are any claims associated with a policy, that information would be stored in the CLAIMS table. In addition a customer can have zero to many dependents such as a child or spouse which would be stored in the DEPENDENT table. All communications with a customer whether through a phone operator, an online chat rep, or the email system will be stored in the COMMUNICATION table.

DBD name: INSURDB | Database access method: (HIDAM,VSAM)

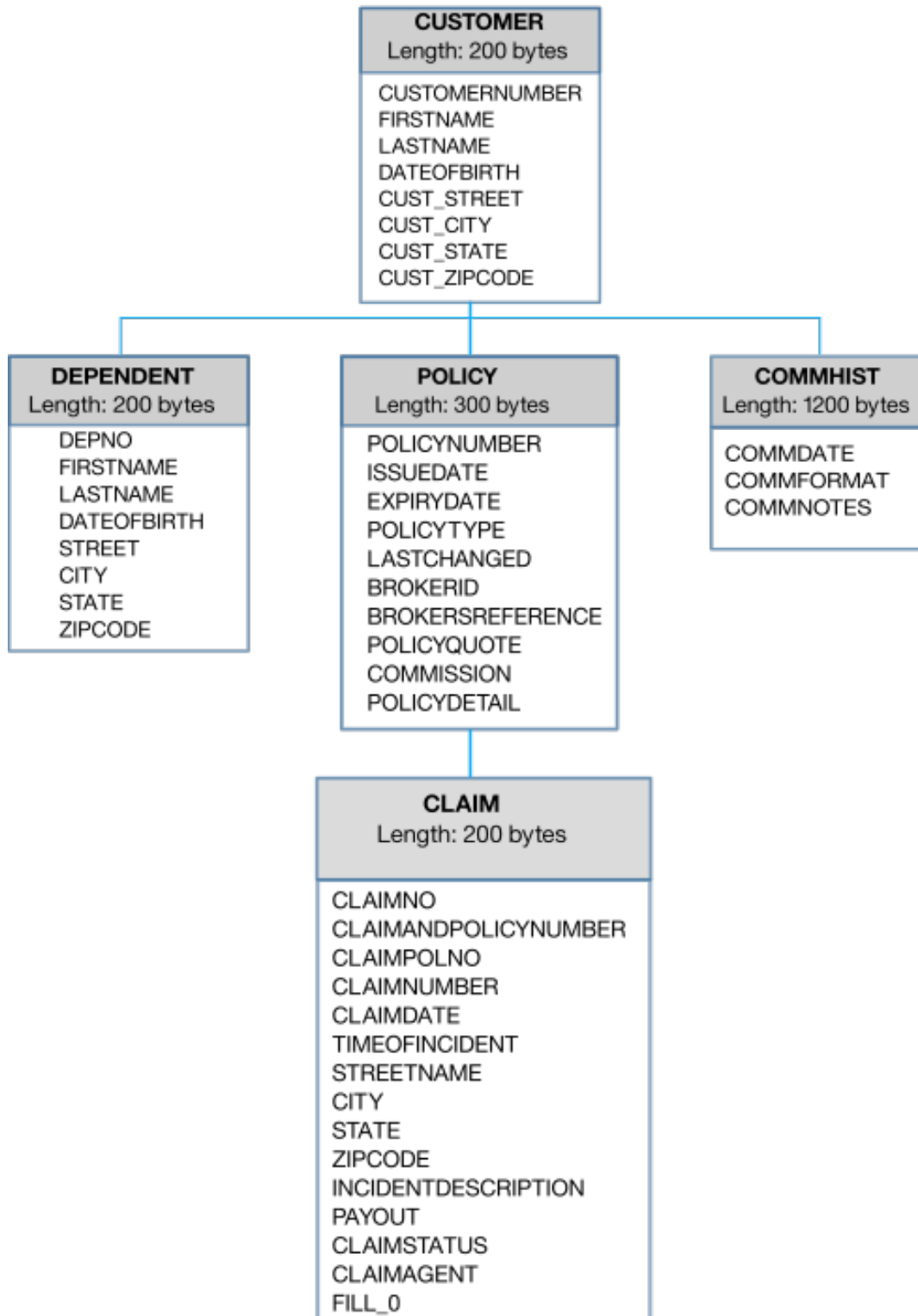


Figure 4. Database structure for the sample IMS Insurance company

Java in IMS solution adoption kit

Modernization helps to keep IMS clients aligned and ready for digital transformation. IMS clients worldwide have a rich portfolio of industry strength applications that are proven and trusted. The Java in IMS solution adoption kit enables users to modernize their IMS applications when the applications are running on z Systems.

Using Java to modernize legacy applications or to develop new business logic enables you to use the skills that are more readily available.

Related information

High-level story that defines the use case scenario

Use case: Java in IMS

The following example shows a fictitious Insurance company as it navigates through and successfully implements the Java in IMS solution.

Goal:

Prosentient Insurance, a fictitious company, is developing a new Claim Handling client portal, which incorporates new Java apps that must access IMS data either by invoking existing IMS transactions or launching new IMS transactions.

Scope

- IMS Database Manager and IMS Transaction Manager are installed and running on z/OS
- Open access to IMS data is implemented. For more information, see [“Implementing open access for Java applications” on page 21.](#)
- IMS Catalog infrastructure is implemented
- The IMS database has been set up for Java application access

Preconditions:

- JMP region is not yet implemented
- Application development environment is not yet set up to support Java

Success End Condition:

- JMP region is established
- Java Developer creates an application that the System Programmer deploys in the JMP
- Application is started and tested

Primary Actors:

- Vikram, System Programmer at Prosentient
- Susan, Application Developer at Prosentient

Secondary Actors:

- Ana, Enterprise Architect at Prosentient
- Lars, an independent industry consultant to Prosentient
- Matt, Project Manager at Prosentient

- Steve, Database Administrator at Prosentient
- Trish, SAF Administrator at Prosentient

Main Use Case Success Scenario:

1. All actors gain a high-level understanding of how Java applications accesses IMS databases
2. Vikram sets up the Java Message Processing (JMP) region in z/OS
3. Susan sets up the Java environment (drivers and adapters) on her machine
4. Susan reviews the database structure and queries
5. Susan develops the Java application
6. Susan deploys the Java application
7. Vikram and Steve confirms system readiness (application, transaction, database and region)
8. Susan runs the Java application
9. Vikram and Susan collaboratively test the Java application, and debug and resolve any problems

Implementing Java in IMS

To get started with a modernizing project with Java in IMS, the first step is to understand just how many different ways Java applications run on z Systems and the software that is required to develop and deploy IMS transactions by using the Java language. From there, you need to ensure that the environment is set up on z/OS to enable Java applications to run in IMS.

About this task

Java can be run in an IMS dependent region or a CICS region or started by a WebSphere Application Server running on z/OS. In this use case, we are running in a Java dependent region.

Application developers must learn how to code the input messages that are required to trigger IMS transactions in Java. When the application is developed and deployed to the IMS environment, the developer can test it by using a sample application that sends the input message to the transaction manager and displays the output message.

This use case depicts a simplistic view of how to modernize IMS by using Java. But, the online transaction processing and database access use cases are more complex. Therefore, this adoption kit should be treated as learning material.

For a complete list of sample code, see [“Samples overview”](#) on page 38.

Procedure

Complete the following steps to learn how to build Java applications for IMS:

1. Review the following resources:
 - [Supercharge IMS business applications with Java](#) (book, in PDF)
 - [Java and IMS](#) (videos)
 - [Introduction to IMS databases \(Database Administration\)](#) (topic)
 - [Defining application program elements for IMS TM \(Application Programming\)](#) (topic)
 - [Java application development for IMS \(Application Programming\)](#) (topic)
 - [IMS solutions for Java development overview \(Application Programming\)](#) (topic)
 - [IMS application development, part 1](#) (video)
 - [IMS application development, part 2](#) (video)
2. Review the following checklists:
 - [Systems programmer checklist](#)

- [Developers checklist](#)
3. Build, run, and test your Java application in IMS, using the following resources as guidelines:
 - [Sample code](#)
 - [Java in IMS demos](#) (video)
 4. Test the application, debug, and resolve any problems, using the following resources as guidelines:
 - [Java and IMS: problem determination](#) (videos)

System Programmer checklist

This checklist is for system programmers who assist Java application developers with their type-2 or local Java application development project in the Java™ Message Processing (JMP) region to access an IMS™ database.

Environment preparation:

1. Install the IMS Java dependent region resource adapter (`imsutm.jar`) and the IMS Universal drivers (`imsudb.jar`) that are provided through the Java On Demand Feature FMID
2. Specify the location for the Java Virtual Machine (JVM) and the IMS Java native code (`libT2DLI.so`). If you use the `//STDENV DD` statement to set Java environment variables and options, modify the shell script that is referenced by the DD statement. Otherwise, modify the DFSJVM EV member in the IMS PROCLIB data set to specify the settings.
3. Specify the location of the .jar files for the IMS Java dependent region resource adapter, the IMS Universal drivers, and the Java applications. If you use the `//STDENV DD` statement to set Java environment variables and options, modify the shell script that is referenced by the DD statement. Otherwise, modify the DFSJVM MS member in the IMS PROCLIB data set to specify the settings.

For each program:

1. Create a new PSB or reuse an existing one (requires `LANG=JAVA` on the PSBGEN macro statement).
2. Create the IMS program to make the PSB available online.
3. Create the IMS transaction and associate it with the IMS program/PSB in the previous step.
4. Modify the DFSJVM AP member in the IMS PROCLIB data set to specify the uppercase Java program name (PSB name) to map the IMS program to the main Java class that will be the main entry point when invoking the transaction.
5. Ensure the Java application and any dependencies (.jar files) are stored in the USS file system location that is defined in either the DFSJVM MS member or the shell script that is referenced by the `//STDENV DD` statement.
6. Tailor the IMS JMP procedure, and start the JMP region.
7. If the IMS transaction or IMS program is stopped, start them.

Java Application Developer checklist

This checklist is for Java application developers who develop type-2 or local Java applications in the Java™ Message Processing (JMP) region to access an IMS™ database.

Environment preparation

1. Obtain the IMS Universal drivers (`imsutm.jar` and `imsudb.jar`) and import them into your project.
2. Ask the System Programmer to set up the environment for running Java applications in the JMP region.
3. Identify the z/OS UNIX System Services (USS) file system location where the application .jar file should be uploaded. The file system location is defined either in the DFSJVM MS member or in the shell script that is referenced by the `//STDENV DD` statement. Obtain the permission to upload files to the identified location.

4. Obtain from the System Programmer and System Administrator the following information that is required to access IMS for testing:
 - IP address or host name
 - Data store name
 - Port number
 - IMS Connect port number
 - RACF ID
 - RACF group name

For each program

1. Define the layout of the input and output messages.
2. Develop the application that accesses the IMS message queue and queries the database.
3. Compile and export the application .jar file. Upload it to the z/OS UNIX System Services (USS) file system in binary mode.
4. Work with the system programmer to ensure that the program, the transaction, and the JMP region are started.

Note: If the application changes, the JMP region might need to be refreshed.

5. Unit test the JMP program with an IMS Connect API client application.

Samples overview

The following sample code pertains to the use case scenario for the insurance company. Use these samples to understand how to implement Java in IMS.

For a complete list of sample code, see: [Insurance company - IMS samples](#)

Sample code for starting the JMP by using Java in IMS

The following code shows sample configurations for starting the JMP procedure by using Java in IMS.

For a complete list of sample code, see [Insurance company - IMS samples](#).

IMS™ provides two ways of configuring Java environment variables and options:

- Configuring the Java environment in the static DFSJVMMS and DFSJVMEV members of the IMS.PROCLIB data set.
- Providing shell scripts that contain Java configurations to an //STDENV DD statement. This method allows dynamic configuration.

The two methods cannot be used at the same time. If an //STDENV DD statement is present, IMS ignores both the DFSJVMEV and DFSJVMMS members of the PROCLIB data set.

The job in the following example references both DFSJVMMS and DFSJVMEV, it specifies the class to be used with the transaction, for example 009, and it specifies where to write out the `System.out` and `System.err` streams.


```

/* JCL FOR THE IMS JAVA REGION (JMP) USED TO EXECUTE YOUR JAVA
/* PROGRAM
/*
/* AUTHOR          LAST CHANGED      PROJECT
/* -----
/* PAUL            04/22/16           INSUR01
/* *****
// EXEC DFSJMP,
//          IMSID=IMDO,JVMOPMAS=DFSJVMMS,ENVIRON=DFSJVMEV,
//          CL1=009,CL2=009,CL3=009,CL4=009
//STEPLIB DD DSN=IMSVS.IMDO.SDFSJLIB,DISP=SHR
//          DD DSN=IMSVS.IMDO.SDFSRESL,DISP=SHR
//          DD DSN=CEE.SCEERUN,DISP=SHR
//          DD DSN=SYS1.CSSLIB,DISP=SHR
//PROCLIB DD DSN=IMSVS.IMDO.PROCLIB,DISP=SHR
//JAVAOUT DD PATH='/tmp/jvm.out',
//          PATHDISP=(KEEP,KEEP),
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=(SIRWXU,SIRWGX,SIRWXO)
//JAVAERR DD PATH='/tmp/jvm.err',
//          PATHDISP=(KEEP,KEEP),
//          PATHOPTS=(OWRONLY,OCREAT,OTRUNC),
//          PATHMODE=(SIRWXU,SIRWGX,SIRWXO)

```

Configuring Java with dynamic scripting

The following sample JCL demonstrates how to use the //STDENV DD statement.

```

//STDENV DD *
# This is a shell script which configures
# environment variables for the Java JVM.
# Variables must be exported to be visible to the launcher.
. /etc/profile
export JAVA_HOME=/usr/lpp/java/J7.0
export PATH=/bin:${JAVA_HOME}/bin
LIBPATH=/lib:/usr/lib:${JAVA_HOME}/bin
export LIBPATH="$LIBPATH":

# Customize your CLASSPATH here.
CLASSPATH="$CLASSPATH":myLibPath/imsudbimsxxx.jar
for i in "${JAVA_HOME}/*.jar"; do
    CLASSPATH="$CLASSPATH":$i
done
# Classpath length can be up to 150K
export CLASSPATH="$CLASSPATH":

# Use this variable to specify the encoding for DD STDOUT and DD STDERR.
export JZOS_OUTPUT_ENCODING=Cp1047
# Use this variable to enable or disable the encoding specified
# in JZOS_OUTPUT_ENCODING.
# The default is true. When the variable is set to false, the encoding
# specified in JZOS_OUTPUT_ENCODING is ignored.
export JZOS_ENABLE_OUTPUT_TRANSCODING=false
...

```

The following JCL provides an example of how to configure Java environment variables and options by using the //STDENV DD statement. You can separate the JVM configurations variables in multiple files and concatenate them in a single //STDENV DD statement. The member names here are provided as examples.

```

//JMP00001 JOB MSGLEVEL=1,MSGCLASS=E,CLASS=K,
//          LINES=999999,TIME=1440,REGION=0M,
//          MEMLIMIT=NOLIMIT
//*
//JMP00001 PROC CL1=001,CL2=000,CL3=000,
//          CL4=000,OPT=W,OVL=0,
//          SPIE=0,TIM=00,VALCK=0,
//          PCB=032,SOD=,STIMER=,
//          NBA=5,OBA=5,IMSID=IMS1,
//          AGN=,SSM=,PREINIT=,
//          ALTID=,PWFI=,APARM=,
//          LOCKMAX=,ENVIRON=,JVMOPMAS=,
//          PARDLI=,PRLD=,JVM=31
//REGION EXEC PGM=DFSRR00,
//          PARM=(JMP,&CL1&CL2&CL3&CL4,
//          &OPT&OVL&SPIE&VALCK&TIM&PCB,&STIMER,&SOD,&NBA,

```

```
//          &OBA,&IMSID,&AGN,&PREINIT,&ALTID,&PWFI,'&APARM',
//          &LOCKMAX,&ENVIRON,,&JVMOPMAS,&PRLD,&SSM,&PARDLI,,
//          &JVM)
//STEPLIB DD DSN=IMSVS.IMDO.SDFSJLIB,DISP=SHR
//          DD DSN=IMSVS.IMDO.SDFSRESL,DISP=SHR
//          DD DSN=CEE.SCEERUN,DISP=SHR
//          DD DSN=SYS1.CSSLIB,DISP=SHR
//STDENV DD DSN=h1q1.IMSCONF(IMS1ENV)
//          DD DSN=h1q2.IMSCONF(IMS1OPT)
//          DD DSN=h1q2.IMSCONF(IMS1MAIN)
//          DD DSN=h1q2.IMSCONF(IMS1DEBUG)
//PRINTDD DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//STDOUT DD SYSOUT=*
//STDERR DD SYSOUT=*
//*
//          PEND
//
//JMP00001 EXEC JMP00001
```

In the example, the *IMS1ENV* file is a shell script that configures the environment variables for the JVM. The *IMS1OPT* file specifies the IBM® JVM runtime options, which are typically prefixed with -X, and Java system properties, which are prefixed with -D. The *IMS1MAIN* file supplies arguments to the Java main method.

```
# IMS1ENV sample file
. /etc/profile
export JAVA_HOME=/usr/lpp/java/J7.0
export PATH=/bin:${JAVA_HOME}/bin
LIBPATH=/lib:/usr/lib:${JAVA_HOME}/bin
export LIBPATH="$LIBPATH":
CLASSPATH="$CLASSPATH":myLibPath/imsudbimsxxxx.jar
for i in ${JAVA_HOME}*.jar; do
    CLASSPATH="$CLASSPATH": "$i"
done
export CLASSPATH="$CLASSPATH":
```

```
# IMS1DEBUG sample file
export DEBUG=Y
```

```
# IMS1OPT sample file
export JZOS_OUTPUT_ENCODING=Cp1047
export JZOS_ENABLE_OUTPUT_TRANSCODING=false
```

```
# IMS1MAIN sample file
# Use this variable to supply arguments to the Java main method
JZOS_MAIN_ARGS="gofast"
```

Sample code for input and output messages

These sample messages reflect the input and output messages that are needed for the Insurance application transaction. The following messages show the type of data that is sent to and returned from IMS.

For a complete list of sample code, see [Insurance company - IMS samples](#)

InputMessage.java

The `InputMessage` class defines the structure of the input message that the Insurance application receives. It defines the field name, field type, the start position, and the length of each field. It also specifies the length of the entire message in the call to the super constructor.

```

package message;

import com.ibm.ims.application.IMSFieldMessage;
import com.ibm.ims.base.DLTypeInfo;

public class InputMessage extends IMSFieldMessage {

    private static final long serialVersionUID = 1L;
    static DLTypeInfo[] fieldInfo =
    {
        //ACTION for our example can be 'get ' or 'put '
        new DLTypeInfo("ACTION", DLTypeInfo.CHAR, 1, 4),
        new DLTypeInfo("CUSTNO", DLTypeInfo.INTEGER, 5, 4),
        new DLTypeInfo("CUST_STREET", DLTypeInfo.CHAR, 9, 25),
        new DLTypeInfo("CUST_CITY", DLTypeInfo.CHAR, 34, 13),
        new DLTypeInfo("CUST_STATE", DLTypeInfo.CHAR, 47, 2),
        new DLTypeInfo("CUST_ZIPCODE", DLTypeInfo.CHAR, 49, 5)

    };

    /**
     * Required no arguments constructor
     */
    public InputMessage()
    {
        super(fieldInfo, 53, false);
    }
}

```

HousePolicyMessage.java

The HousePolicyMessage class defines the output message structure for the house policy segment. It defines the field name, field type, the start position, and the length of each field. It also specifies the length of the entire message in the call to the super constructor.

```

package message;

import com.ibm.ims.application.IMSFieldMessage;
import com.ibm.ims.base.DLTypeInfo;

public class HousePolicyMessage extends IMSFieldMessage {
    private static final long serialVersionUID = 23432884;

    static DLTypeInfo[] fieldInfo = {

        new DLTypeInfo("POLICYNUMBER", DLTypeInfo.INTEGER, 1, 4),
        new DLTypeInfo("HOMEPROPERTYTYPE", DLTypeInfo.CHAR, 5, 15),
        new DLTypeInfo("HOMEBEDROOMS", DLTypeInfo.SMALLINT, 20, 2),
        new DLTypeInfo("HOMEHOUSEVALUE", DLTypeInfo.INTEGER, 22, 4),
        new DLTypeInfo("HOME_STREET", DLTypeInfo.CHAR, 26, 25),
        new DLTypeInfo("HOME_ZIPCODE", DLTypeInfo.CHAR, 51, 5)

    };

    public HousePolicyMessage() {
        super(fieldInfo, 55, false);
    }
}

```

CustomerOutputMessage.java

The CustomerOutputMessage class defines the customer output message. This code represents the output message structure for the customer segment. It defines the field name, field type, the start position, and the length of each field. It also specifies the length of the entire message in the call to the super constructor.

This message has additional customized fields to pass back information to the client, for example, fields for the number of house policies, number of auto policies, and an error message.

```

package message;

import com.ibm.ims.application.IMSFieldMessage;
import com.ibm.ims.base.DLTypeInfo;

public class CustomerOutputMessage extends IMSFieldMessage {
    private static final long serialVersionUID = 23432884;

    static DLTypeInfo[] fieldInfo = {

        new DLTypeInfo("CUSTOMERNUMBER", DLTypeInfo.INTEGER, 1, 4),
        new DLTypeInfo("FIRSTNAME", DLTypeInfo.CHAR, 5, 10),
        new DLTypeInfo("LASTNAME", DLTypeInfo.CHAR, 15, 20),
        new DLTypeInfo("DATEOFBIRTH", DLTypeInfo.CHAR, 35, 10),
        new DLTypeInfo("CUST_STREET", DLTypeInfo.CHAR, 45, 25),
        new DLTypeInfo("CUST_CITY", DLTypeInfo.CHAR, 70, 13),
        new DLTypeInfo("CUST_STATE", DLTypeInfo.CHAR, 83, 2),
        new DLTypeInfo("CUST_ZIPCODE", DLTypeInfo.CHAR, 85, 5),
        new DLTypeInfo("NUM_HOUSE", DLTypeInfo.INTEGER, 90, 4),
        new DLTypeInfo("NUM_AUTO", DLTypeInfo.INTEGER, 94, 4),
        new DLTypeInfo("ErrorMessage", DLTypeInfo.CHAR, 98, 500)

    };

    public CustomerOutputMessage() {
        super(fieldInfo, 597, false);
    }
}

```

AutoPolicyMessage.java

The AutoPolicyMessage class defines the output message structure for the auto policy segment. It defines the field name, field type, the start position, and the length of each field. It also specifies the length of the entire message in the call to the super constructor.

```

package message;

import com.ibm.ims.application.IMSFieldMessage;
import com.ibm.ims.base.DLTypeInfo;

public class AutoPolicyMessage extends IMSFieldMessage {
    private static final long serialVersionUID = 23432884;

    static DLTypeInfo[] fieldInfo = {

        new DLTypeInfo("POLICYNUMBER", DLTypeInfo.INTEGER, 1, 4),
        new DLTypeInfo("CAR_MAKE", DLTypeInfo.CHAR, 5, 15),
        new DLTypeInfo("CAR_MODEL", DLTypeInfo.CHAR, 20, 15),
        new DLTypeInfo("CAR_MANUFACTUREDATE", DLTypeInfo.CHAR, 35, 10),
        new DLTypeInfo("CAR_REGNUMBER", DLTypeInfo.CHAR, 45, 7),
        new DLTypeInfo("CAR_DRIVERNAME", DLTypeInfo.CHAR, 52, 30)

    };

    public AutoPolicyMessage() {
        super(fieldInfo, 81, false);
    }
}

```

Sample code for DFSJVMAP

The following code shows the mapping of the PSB names to the Java class entry point.

For a complete list of sample code, see: [Insurance company - IMS samples](#).

```

* This is a mapping of PSB names to Java applications.
* The location of the specific file must be specified separately
* by the DFSJVMMS member in the shareable application classpath.
*
* PSB           Regions   Java programs
* -----
* INSUR01       JMP       controller/Insurance
*****
INSUR01=controller/Insurance

```

Note: You can also use the //STDENV DD statement to dynamically specify the JVM options in shell scripts. If an //STDENV DD statement is present, the DFSJVMAP member is ignored by the IMS system.

Sample code for DFSJVMMS

The following code specifies the classpath and JVM options.

The DFSJVMMS member of the IMS™ PROCLIB data set to specify JVM options for the standalone JVM for JBP regions. This sample demonstrates how the fully qualified class path is specified by using the -Djava.class.path option. The " :>" notation in the class path is a continuation marker for a given Java system property or option.

Note: If you have IMS 14 APAR PI68127 applied, you can also use the //STDENV DD statement to dynamically specify the JVM options in shell scripts. If an //STDENV DD statement is present, the DFSJVMMS member is ignored by the IMS system.

```

*****
* Specify the profile that has environment settings and JVM options.
* Or specify the class path, as demonstrated in this example.
*****
-Djava.class.path=>
/u/ims/usr/lpp/ims/ims13/imsjava/imsudb.jar:>
/u/ims/usr/lpp/ims/ims13/imsjava/imsutm.jar:>
/path/to/yourJavaApp.jar
*****
* The following JVM options are a subset of the options allowed
* Xmaxf0.8   - specifies the maximum percentage of heap that must be
*             free after a garbage collection
* Xmx64M     - set the maximum Java heap size of your program
* Xms0512k   - set stack size for OS threads for 32 bit
* Xss256k    - set Java thread stack size
* Xms1M      - set initial Java heap size
*****
-Xmaxf0.8
-Xminf0.3
-Xmx32M
-Xms0512k
-Xss256k
-Xms1M

```

For a complete list of sample code, see: [Insurance company - IMS samples](#).

Sample code for DFSJVMEV

The following code specifies the LIBPATH, including the path to the JVM, and path to the IMS Java native code.

For a complete list of sample code, see [Insurance company - IMS samples](#).

```

* Specify the location of Java Virtual Machine (JVM) installation and
* IMS Java native code (libT2DLI.so)
*****
LIBPATH=>
/java7/J7.1/bin/j9vm:>
/java7/J7.1/bin:>
/u/ims/usr/lpp/ims/ims13/imsjava/lib
*

```

Note: You can also use the //STDENV DD statement to dynamically specify the JVM options in shell scripts. If an //STDENV DD statement is present, the DFSJVMENV member is ignored by the IMS system.

Sample application code for our use case

This sample application code retrieves the input message from the message queue, extracts the data from the input message, queries the database, and returns the output.

For a complete list of sample code, see: [Insurance company - IMS samples](#)

```
package controller;

import java.sql.Connection;
import java.util.List;

import com.ibm.ims.dli.DLIException;
import com.ibm.ims.dli.tm.Application;
import com.ibm.ims.dli.tm.ApplicationFactory;
import com.ibm.ims.dli.tm.IOMessage;
import com.ibm.ims.dli.tm.MessageQueue;
import com.ibm.ims.dli.tm.Transaction;
import com.ibm.ims.jdbc.IMSDataSource;

import customer.info.Customer;
import customer.info.CustomerService;
import policy.info.AutoPolicy;
import policy.info.HousePolicy;

public class Insurance {

    public static void main(String[] args) {

        System.out.println("Controller called");

        CustomerService customerService = new CustomerService();
        Connection conn = null;

        //Application is used to get a Transaction object
        Application app = ApplicationFactory.createApplication();

        //Transaction is primarily used for commit or rollback calls
        Transaction tran = app.getTransaction();

        //Get a handle to the MessageQueue object for sending and receiving
        //messages to and from the IMS message queue
        MessageQueue messageQueue = app.getMessageQueue();

        IOMessage inputMessage = null;
        IOMessage customerOutputMessage = null;
        IOMessage autoPolicyMessage = null;
        IOMessage housePolicyMessage = null;

        try {

            //initialize the input and output messages to the IOMessage object
            inputMessage = app.getIOMessage("class://message.InputMessage");
            customerOutputMessage = app.getIOMessage("class://
message.CustomerOutputMessage");
            autoPolicyMessage = app.getIOMessage("class://
message.AutoPolicyMessage");
            housePolicyMessage = app.getIOMessage("class://
message.HousePolicyMessage");

            //get a InputMessage message from the queue, if there is one
            while (messageQueue.getUnique(inputMessage)) {

                //get the customer number from the InputMessage
                int customerNumber = inputMessage.getInt("CUSTNO");

                //get the action from the InputMessage
                String action = inputMessage.getString("ACTION");

                //connect to the database
                IMSDataSource dataSource = new IMSDataSource();
                dataSource.setDriverType(IMSDatasource.DRIVER_TYPE_2);
                dataSource.setDatastoreName("IMD0");
```

```

        dataSource.setDatabaseName("INSUR01");
        conn = dataSource.getConnection();
        conn.setAutoCommit(false);

        //update customer information
        if(action.trim().equalsIgnoreCase("put")) {

            customerOutputMessage.setString("ErrorMessage",
                "Put operation not yet implemented");
            messageQueue.insert(customerOutputMessage,
MessageQueue.DEFAULT_DESTINATION);

        }

        //get customer information
        if(action.trim().equalsIgnoreCase("get")) {

            //query the database passing in the customer number,
            //if getConnection() gets an exception this code is
            //not reached
            Customer customerInfo =
customerService.getCustomerInfo(customerNumber, conn);

            //if a customer is found then return the information
            if(customerInfo != null) {

                int numAutoPolicy =
customerInfo.getAutoPolicies().size();
                int numHousePolicy =
customerInfo.getHousePolicies().size();

                customerOutputMessage.setString("FIRSTNAME",
customerInfo.getFirstName());
                customerOutputMessage.setString("LASTNAME",
customerInfo.getLastName());
                customerOutputMessage.setString("CUST_STREET",
customerInfo.getStreet());
                customerOutputMessage.setString("CUST_CITY",
customerInfo.getCity());
                customerOutputMessage.setString("CUST_STATE",
customerInfo.getState());
                customerOutputMessage.setString("CUST_ZIPCODE",
customerInfo.getZipCode());
                customerOutputMessage.setString("DATEOFBIRTH",
customerInfo.getDateOfBirth());

                //set the number of auto and house policies
                customerOutputMessage.setInt("NUM_AUTO",
numAutoPolicy);
                customerOutputMessage.setInt("NUM_HOUSE",
numHousePolicy);

                //insert the populated CustomerOutputMessage
                messageQueue.insert(customerOutputMessage,
MessageQueue.DEFAULT_DESTINATION);

                //loop thru the list of auto policies for the
                //populate the AutoPolicyMessage and insert
                List<AutoPolicy> autoPolicies =
customerInfo.getAutoPolicies();
                for(AutoPolicy autoPolicy : autoPolicies) {

                    autoPolicyMessage.setInt("POLICYNUMBER",
autoPolicy.getPolicyNumber());
                    autoPolicyMessage.setString("CAR_MAKE",
autoPolicy.getMake());
                    autoPolicyMessage.setString("CAR_MODEL",
autoPolicy.getModel());
                    autoPolicyMessage.setString("CAR_MANUFACTUREDATE",
autoPolicy.getManufactureDate());
                    autoPolicyMessage.setString("CAR_REGNUMBER",
autoPolicy.getRegistrationNumber());
                    autoPolicyMessage.setString("CAR_DRIVERNAME",
autoPolicy.getDriverName());

                    //insert each AutoPolicyMessage into
                    queue. Each message is a segment.

```

```

MessageQueue.DEFAULT_DESTINATION);
    }

    //loop thru the list of house policies for the
    //populate the HousePolicyMessage and insert
    List<HousePolicy> housePolicies =
    customerInfo.getHousePolicies();
    for(HousePolicy housePolicy : housePolicies) {

        housePolicyMessage.setInt("POLICYNUMBER", housePolicy.getPolicyNumber());
        housePolicyMessage.setShort("HOMEBEDROOMS", housePolicy.getNumberOfBedrooms());
        housePolicyMessage.setInt("HOMEHOUSEVALUE", housePolicy.getValue());
        housePolicyMessage.setString("HOMEPROPERTYTYPE", housePolicy.getPropertyType());
        housePolicyMessage.setString("HOME_STREET", housePolicy.getStreet());
        housePolicyMessage.setString("HOME_ZIPCODE", housePolicy.getZipCode());

        //insert each HousePolicyMessage into
        queue. Each message is a segment.
        messageQueue.insert(housePolicyMessage, MessageQueue.DEFAULT_DESTINATION);
    }

    conn.close();
    tran.commit();

    } //end if customer not null
    //if no customer is found
    else {

        customerOutputMessage.setString("ErrorMessage",
                                        "No Customer has been found
        with a customer number of " + customerNumber);
        messageQueue.insert(customerOutputMessage,
        MessageQueue.DEFAULT_DESTINATION);
    }

    if action=get
        } //end
    } //end getUnique while loop
    } catch (Exception e) {
        System.out.println(e.getMessage());
        try {
            conn.close();
        } catch (Exception e1) {
            System.out.println(e1.getMessage());
        }
        try {
            if(e.getMessage().length()>500){
                customerOutputMessage.setString("ErrorMessage",
                e.getMessage().substring(0 , 500));
            } else {
                customerOutputMessage.setString("ErrorMessage",
                e.getMessage());
            }
            messageQueue.insert(customerOutputMessage,
            MessageQueue.DEFAULT_DESTINATION);
        } catch (DLIException e1) {
            System.out.println("DLIException encountered");
            System.out.println(e1.getMessage());
        }
    }

```



```

        }
        finally {
            app.end();
        }
    }
}

```

Sample IMS Connect based client application for testing Java transactions

This sample testing application simply tests the application code for our use case.

For a complete list of sample code, see: [Insurance company - IMS samples](#)

```

package client;

import java.io.ByteArrayOutputStream;

import java.io.IOException;
import java.io.UnsupportedEncodingException;
import java.util.Vector;

import com.ibm.ims.connect.ApiProperties;
import com.ibm.ims.connect.Connection;
import com.ibm.ims.connect.ConnectionFactory;
import com.ibm.ims.connect.ImsConnectApiException;
import com.ibm.ims.connect.InputMessage;
import com.ibm.ims.connect.OutputMessage;
import com.ibm.ims.connect.TmInteraction;
import com.ibm.ims.dli.conversion.util.ConversionUtils;

public class Client {

    String host, datastore, tranName, racfId, racfGroupName, clientId;
    int port;
    private static final String CODEPAGE = "cp037";
    static Vector<byte[]> byteArray = new Vector<byte[]>();

    public static void main(String[] args) throws Exception {

        Client client = new Client("INSUR01", "x.xx.xxx.xxx", "IMD0", 8890, "PAUL",
"SYS1", "CLIENX1");

        //create input message
        //ACTION
        client.appendString("get", 4);

        //CUSTNO
        client.appendInt(2);

        //CUST_STREET
        client.appendString("Rustic Place", 25);

        //CUST_CITY
        client.appendString("Palo Alto", 13);

        //CUST_STATE
        client.appendString("CA", 2);

        //CUST_ZIPCODE
        client.appendString("95002", 5);

        client.invoke();

    }

    /**
     * Pad the string to reach the desired length
     * @param value
     * @param length
     * @return
     */
    public String padRight(String value, int length) {

```

```

        if(value.length() > length) {
            String shortenedString = value.substring(0, length);
            return shortenedString;
        }
        return String.format("%1$-" + length + "s", value);
    }

    /**
     *
     * @param tranName 8 character transaction name
     * @param host the IP address of the z/OS system
     * @param dataStore the IMS dataStore name
     * @param port the IMS Connect port
     * @param racfId 8 character RACFID
     * @param racfGroupName 8 character RACF GROUP ID
     * @param clientId 8 character IMS Connect client ID
     */
    public Client(String tranName, String host, String dataStore, int port, String racfId,
        String racfGroupName, String clientId) {

        this.tranName = tranName;
        this.host = host;
        this.port = port;
        this.dataStore = dataStore;
        this.racfId = racfId;
        this.racfGroupName = racfGroupName;
        this.clientId = clientId;
    }

    /**
     * Invoke the transaction
     * @throws ImsConnectApiException
     * @throws Exception
     */
    public void invoke() throws ImsConnectApiException, Exception {

        Connection myConnection = null;
        TmInteraction myTmInteraction = null;
        ConnectionFactory myConnectionFactory = new ConnectionFactory();
        myConnectionFactory.setHostName(host);
        myConnectionFactory.setPortNumber(port);
        myConnectionFactory.setUseSslConnection(false);
        myConnection = myConnectionFactory.getConnection();
        myConnection.setClientId(clientId);
        myTmInteraction = myConnection.createInteraction();

        //trancode with length 8
        myTmInteraction.setTrancode(tranName);
        myTmInteraction.setImsDataStoreName(dataStore);
        myTmInteraction.setRacfUserId(racfId);
        myTmInteraction.setRacfGroupName(racfGroupName);

        myTmInteraction.setInputMessageDataSegmentsIncludeLlzzAndTrancode(ApiProperties.INPUT_MESSAGE_DATA_SEGMENTS_DO_NOT_INCLUDE_LLZZ_AND_TRANCODE);
        myTmInteraction.setImsConnectTimeout(ApiProperties.TIMEOUT_2_SECONDS);
        myTmInteraction.setInteractionTimeout(ApiProperties.TIMEOUT_2_SECONDS);

        myTmInteraction.setInteractionTypeDescription(ApiProperties.INTERACTION_TYPE_DESC_SENDRECV);
        myTmInteraction.setImsConnectCodepage(CODEPAGE);

        // get InputMessage instance from myTmInteraction
        InputMessage imsConnectInputMessage = myTmInteraction.getInputMessage();

        // Populate InputMessage object with input byte array
        imsConnectInputMessage.setInputMessageData(getByteMessage());

        // execute the transaction
        myTmInteraction.execute();

        // get output from myTmInteraction
        OutputMessage outputMessage = myTmInteraction.getOutputMessage();

        // get data from outMsg as a string
        displayData(outputMessage.getDataAsString().replace((char)0x00, ' '),
        outputMessage);
    }

```

```

        myConnection.disconnect();
    }

    /**
     * Convert an int to a byte array and add to Vector of byte arrays
     * @param value
     */
    public void appendInt(int value) {
        byte[]
intBytes = ConversionUtils.convertToByte(value);
        byteArray.addElement(intBytes);
    }

    public void appendShort(short value) {
        byte[]
shortBytes = ConversionUtils.convertToByte(value);
        byteArray.addElement(shortBytes);
    }

    /**
     * Convert String to byte array and add to Vector of byte arrays. String will be
padded to the length specified
     * @param value the String to be converted
     * @param length the length of the padded String
     */
    public void appendString(String value, int length) {
        byte stringBytes[];

        //pad the data so it meets the length of the field
        String paddedString = padRight(value, length);

        try {
            stringBytes = paddedString.getBytes(CODEPAGE);
            byteArray.addElement(stringBytes);
        } catch (UnsupportedEncodingException e) {
            System.out.println(e.getMessage());
        }
    }

    public void appendString(String value) {
        byte stringBytes[];

        try {
            stringBytes = value.getBytes(CODEPAGE);
            byteArray.addElement(stringBytes);
        } catch (UnsupportedEncodingException e) {
            System.out.println(e.getMessage());
        }
    }

    /**
     * Creates a byte array from a Vector of byte array elements.
     * The byte array will be sent to the specified IMS transaction.
     * @return byte array
     */
    public byte[]
getBytesMessage() {
        ByteArrayOutputStream outputStream = new ByteArrayOutputStream();

        for(byte[] arrayElement : byteArray) {
            try {
                outputStream.write(arrayElement );
            } catch (IOException e) {
                System.out.println(e.getMessage());
            }
        }
    }

```

```

    }
    return outputStream.toByteArray();
}

private void displayData(String outputData, OutputMessage outputMessage) {
    byte [][] arrayOfData = outputMessage.getDataAsArrayOfByteArrays();
    for (int i = 0; i < arrayOfData.length; i++) {
        String s=null;
        try {
            s = new String(arrayOfData[i], CODEPAGE);
        } catch (UnsupportedEncodingException e) {

            System.out.println(e.getMessage());
        }

        System.out.println(s);
    }
    System.out.println("Done");
}
}
}

```

Sample architectural diagram

The following figure shows a Java application that is running in a JMP or JBP region.

Database access and message processing requests are passed to the IMS Java dependent region resource adapter and type-2 IMS Universal drivers, which converts the calls to DL/I calls.

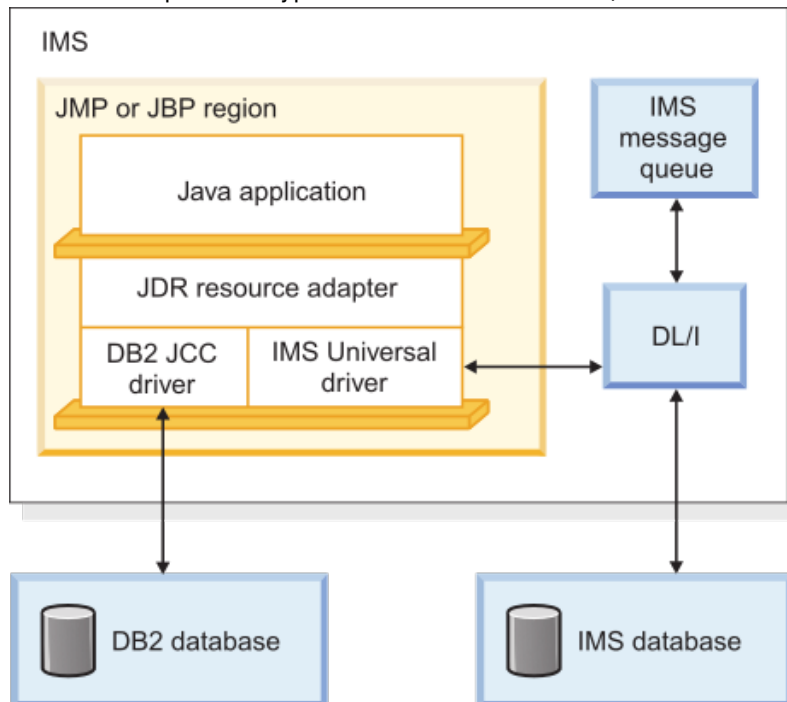


Figure 5. JMP or JBP application that is using the IMS Java dependent region resource adapter

Sample z/OSMF workflow

The IMS System Programmer uses the z/OSMF workflow to automate the set up of the z/OS environment for the Java application and dependent region. The workflow creates the PROCLIB dataset members, the JMP startup procedure, the transaction, the program, and starts all of these resources.



Attention: The z/OSMF workflow is designed to be a sample. Therefore, you may need to make changes in order to adapt the workflow to your environment.

To view the sample z/OSMF workflow; the .xml definition file for Java set up, see [the z/OSMF sample](#).

For more information about z/OSMF, see [IBM z/OS Management Facility Online Help](#).

Sample database structure

The following figure depicts the IMS database layout for the sample Insurance company for Java in IMS.

The CUSTOMER table contains information about the insurance company customer. Each customer can have many policies, which are stored in the POLICY table. If there are any claims that are associated with a policy, that information would be stored in the CLAIMS table. In addition, a customer can have zero to many dependents such as a child or spouse, which would be stored in the DEPENDENT table. All communications with a customer whether through a phone operator, an online chat rep, or the email system is stored in the COMMUNICATION table.

DBD name: INSURDB | Database access method: (HIDAM,VSAM)

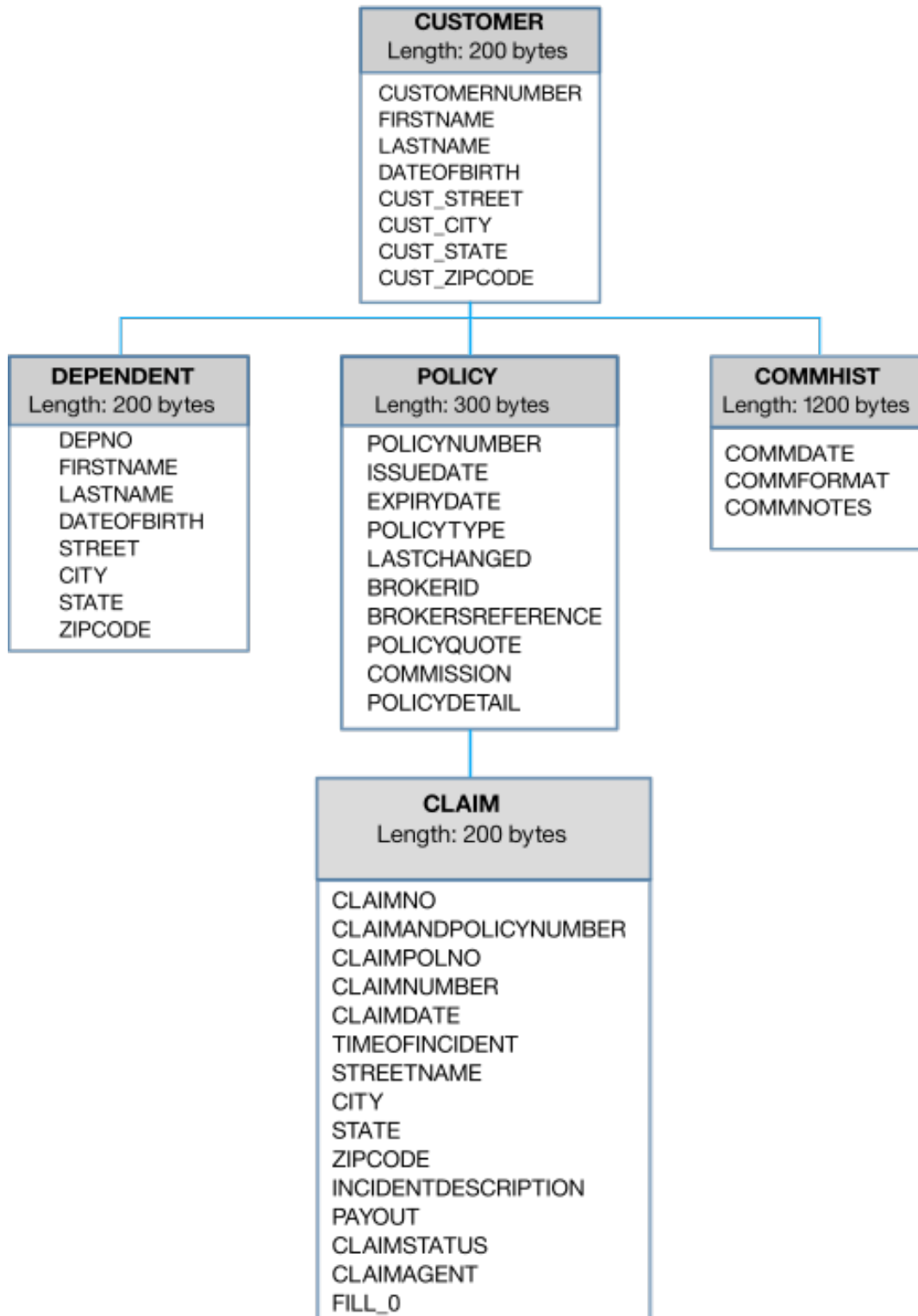


Figure 6. Database structure for the sample IMS Insurance company

API economy solution adoption kit

In the API economy, application programming interfaces (APIs) are the digital glue that links services, applications, and systems. The API economy solution adoption kit provides the information to guide you through the process to integrate your IMS assets into the overall API solution.

You can enable IMS applications as REST services and define REST APIs for these services for IMS assets to be part of the API economy. z/OS® Connect Enterprise Edition (z/OS Connect EE) V2.0 provides a REST endpoint for mainframe assets, and the IMS service provider that is included in z/OS Connect EE V2.0.5 (APAR PI70432) and later provides API access to IMS.

Important: The information and implementation workflow provided in this kit is based on z/OS Connect EE Version 2.0, where IMS services are created by using IMS Explorer, and REST APIs are created by using z/OS Connect EE API Editor. In z/OS Connect EE V3.0, service creation and API creation functions are consolidated into the new z/OS Connect EE API toolkit.

Use case: API economy

The following example shows a fictitious Insurance company as it navigates through and successfully implements the API economy solution for IMS.

Goal:

Prosentient Insurance, a fictitious company, needs to enable different regional teams to develop their localized car insurance approval applications based on local regulations. All these applications need to access IMS data through an existing IMS application. Because these insurance approval applications must be light-weight and support mobile devices, a REST API is required for these application developers.

Preconditions:

No existing REST solutions for accessing IMS transactions.

Success End Condition:

A REST API is available for mobile application developers to develop their applications to access the IMS application (dream car).

Primary Actors:

- Dan, System Administrator at Prosentient
- Bob, Application Programmer at Prosentient
- Shavon, API Developer at Prosentient

Secondary Actors:

- Ana, Enterprise Architect at Prosentient
- Trish, SAF Administrator at Prosentient
- Andre, Mobile App Developer at Prosentient, North America region

Main Use Case Success Scenario:

The following workflow is based on z/OS Connect EE Version 2.0, where service creation is done in IMS Explorer V3.2, and API creation is done in z/OS Connect EE V2 API Editor.

1. Each of the roles in the project familiarizes themselves with the overall process, tools, and skills that are involved to implement the solution, with particular focus on the areas for which they have responsibility.
2. Dan installs z/OS Connect EE V2.0 and configure to use the IMS service provider, with help from Trish for security configuration.
3. Bob downloads and installs the IMS Explorer.
4. Bob creates a mobile (REST/JSON) service from the IMS dream car (insurance approval) application in IMS Explorer V3.2.
5. Bob creates a test case to test the mobile service in IMS Explorer.
6. Shavon downloads and installs z/OS Connect EE API Editor.
7. Shavon develops a REST API for the mobile service that accesses the IMS dream car application in z/OS Connect EE API Editor.
8. Shavon deploys and tests the REST API in z/OS Connect EE API Editor.
9. Dan promotes the mobile service to the production server.
10. Dan deploys the REST API to the production server.
11. Dan uses IBM API Connect to secure and publish the APIs to a developer portal and to manage API subscriptions.
12. Andre discovers this new API on the developer portal and proceeds to develop his application based on the API.

Note: For z/OS Connect EE V3.0, the support for service creation and API creation is consolidated into one tool, the z/OS Connect EE V3 API toolkit.

Implementing API economy solution in IMS

To get started, the first step is to understand the business requirements, the IMS transactions, transaction messages, and data that need to take part in the API economy, and the personnel, tools, and products that need to be involved in implementing the solution. Then you can involve the appropriate members on your team to install, configure, design, and implement the solution.

About this task

Important: The following information is based on z/OS Connect EE Version 2.0. For the steps for z/OS Connect EE Version 3.0 to create an IMS and an API to access the service, see the following scenarios in z/OS Connect EE V3.0 documentation:

- [Create an IMS service](#)
- [Create an API to invoke an IMS service](#)

Table 3. Implementing the API economy solution for IMS		
Steps	Procedure	User role who completes the task
1	Learn and research about API economy and how IMS assets can be enabled as part of the API economy. • z/OS Connect and the API economy (IBM Mainframe Development site)	Everyone
2	Install z/OS Connect. (topic)	Dan, System administrator
3	Using the IMS service provider. (topic)	
4	Install IMS Explorer. (topic)	Bob, Application programmer

Table 3. Implementing the API economy solution for IMS (continued)

Steps	Procedure	User role who completes the task
5	Enable an IMS transaction as a REST service. In IMS Explorer: 1. Connect to the z/OS Connect EE server (topic)	Bob, Application programmer
6	Install the z/OS Connect EE API Editor . (topic)	Shavon, API developer
8	Deploy and test the REST API . (topic)	Shavon, API developer
9	Deploy a service to the production server . (IBM technote with sample scripts)	Dan, System administrator
10	How to manage services . (topic)	Dan, System administrator
11	Secure and publish the APIs to a developer portal and manage the API subscriptions by using IBM API Connect®. See the IBM Redpaper publication: <ul style="list-style-type: none"> • Getting Started with IBM API Connect: Concepts and Architecture Guide (IBM Redpaper) • Getting Started with IBM API Connect: Scenarios Guide (IBM Redpaper) 	Shavon and Dan

Index

A

access kit
 open [5–7](#), [10](#), [18](#), [20–22](#), [25](#), [33](#)
adoption
 kit [1](#), [5–7](#), [10](#), [14](#), [18](#), [20–23](#), [25](#), [26](#), [29](#), [33](#), [35–38](#), [40](#),
 [42–44](#), [47](#), [50](#), [51](#), [53](#), [54](#)
API economy
 implementation [54](#)
 IMS [53](#)
 modernization [54](#)
 REST [53](#)
 use case [53](#)
APIs
 modernization [53](#)
application development
 artificial intelligence(AI) [4](#)
artificial intelligence(AI)
 application development [4](#)
 IMS Transaction Manager [4](#)
 transaction management [4](#)

C

code
 sample [7](#), [10](#), [14](#), [22](#), [23](#), [26](#), [29](#), [38](#), [40](#), [42–44](#), [47](#), [51](#)

I

IMS
 API economy [53](#)
 Java [1](#), [35](#), [50](#), [51](#)
 Java kit [38](#)
 REST services [53](#)
IMS Transaction Manager)
 artificial intelligence(AI) [4](#)
interactive
 model [5](#), [10](#), [18](#), [20](#), [25](#), [33](#)

J

Java
 IMS [1](#), [35](#), [38](#), [50](#), [51](#)
 modernization [35–37](#)
 use case [1](#), [35](#), [50](#), [51](#)

K

kit
 adoption [1](#), [5–7](#), [10](#), [14](#), [18](#), [20–23](#), [25](#), [26](#), [29](#), [33](#),
 [35–38](#), [40](#), [42–44](#), [47](#), [50](#), [51](#), [53](#), [54](#)

M

model
 interactive [5](#), [20](#)

modernization
 API economy [54](#)
 Java [35–37](#)
 REST APIs [53](#), [54](#)

O

open
 access kit [5–7](#), [10](#), [18](#), [20–22](#), [25](#), [33](#)
overview
 samples [7](#), [22](#), [38](#)

S

sample
 code [7](#), [10](#), [14](#), [22](#), [23](#), [26](#), [29](#), [38](#), [40](#), [42–44](#), [47](#), [51](#)
samples
 overview [7](#), [22](#), [38](#)

T

transaction management
 artificial intelligence(AI) [4](#)

U

use case
 API economy [53](#)
 Java [1](#), [35](#), [50](#), [51](#)

