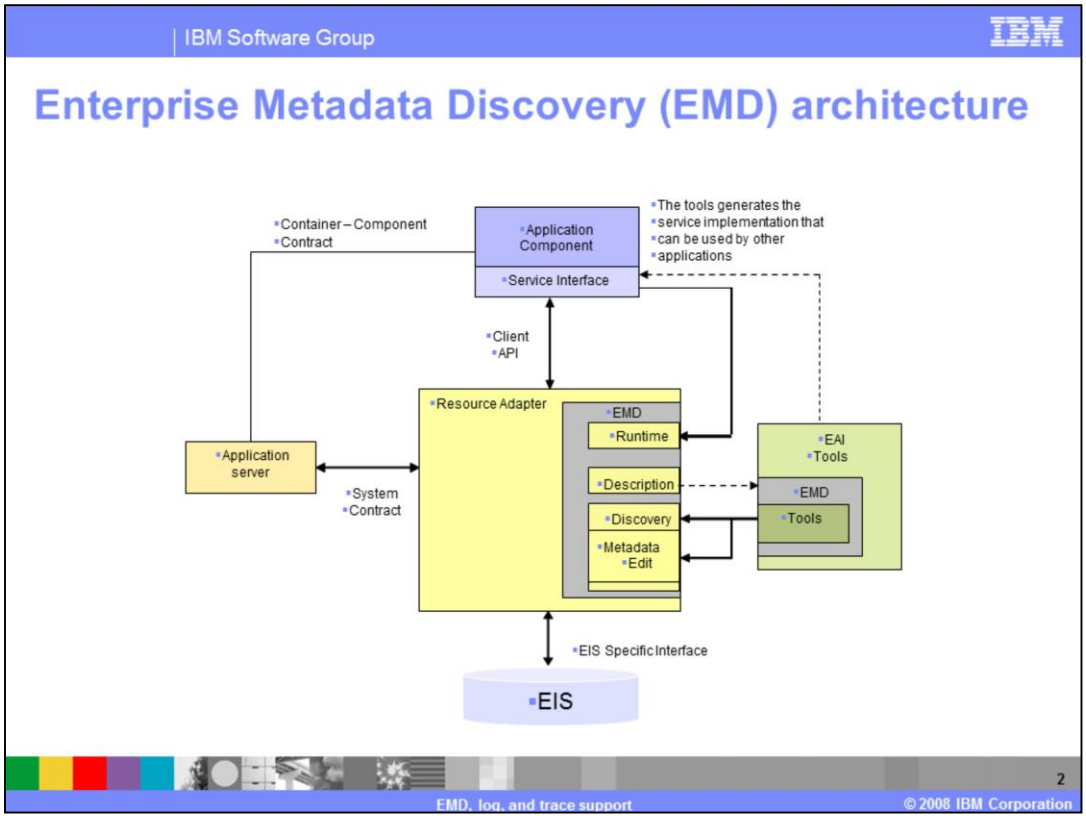
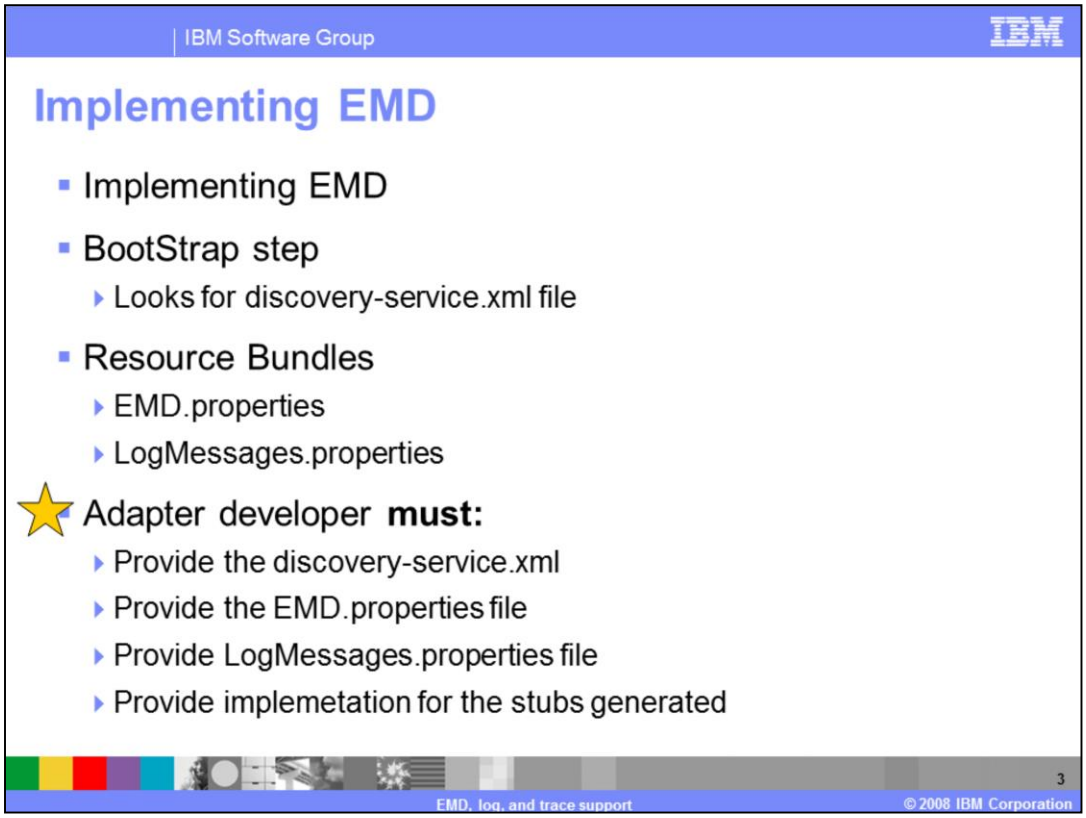
This presentation covers the high level details on the classes and the methods that need to be implemented for metadata discovery, logging and tracing support in your custom WebSphere® adapter.



The slide shows an architectural view of Enterprise Metadata Discovery(EMD). The major portion of EMD is the tools contract, which specifies how the adapter can introspect the EIS and provide the necessary information to build a service. The EMD contract also has runtime contracts. The J2CA specification was designed before the introduction of SOA, so there are a few pieces necessary to bridge the gap between a “service” and a JCA connector. These are discussed later on in the presentation.



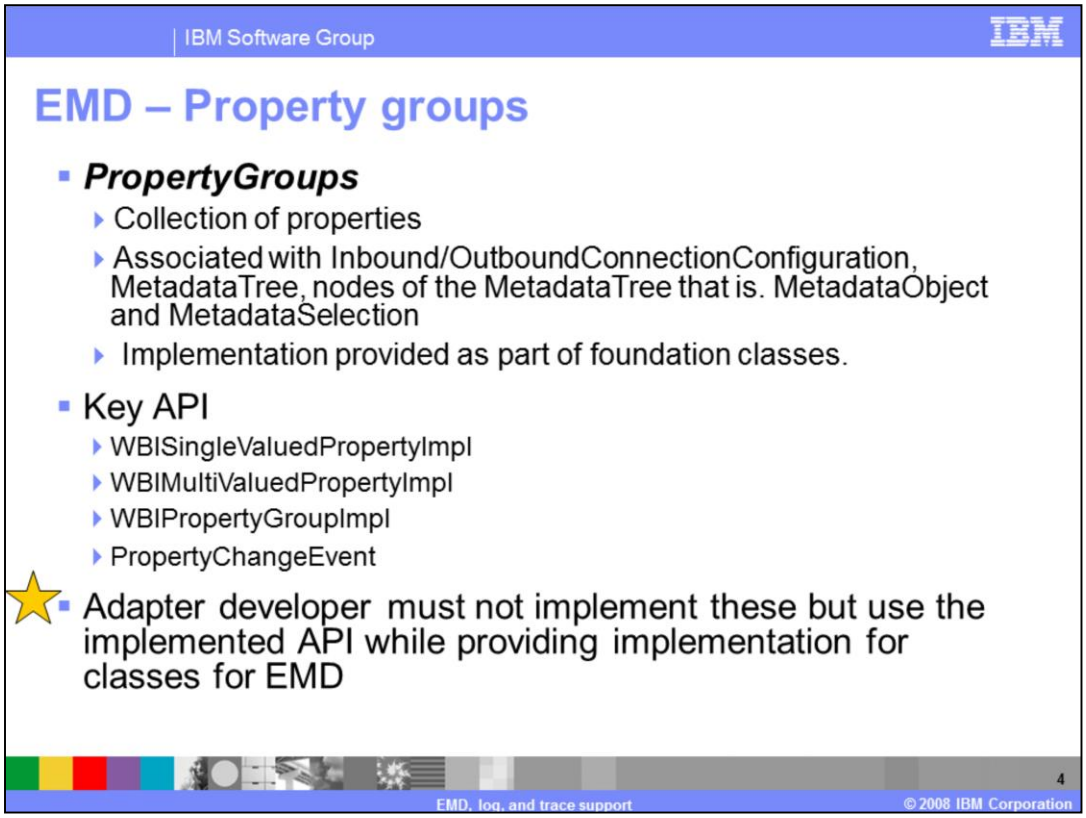
Enterprise Metadata Discovery (EMD) is a discovery service component within an adapter that enables the generation of business object definitions and other artifacts required by SCA. It is equivalent to the Object Discovery Agent (ODA) from WebSphere Business Integration adapters. In addition to generating business objects, however, it also generates the artifacts needed by SCA, such as import, export files and WSDL.

For the EMD tool to be able to recognize the resource adapter enabled for EMD, there is a bootstrap step that looks for a discovery-service.xml file. This file should be located in the META-INF folder for the resource adapter. The structure and content of this file can be referred from EMD specification. The file is provided as part of Twine ball sample which can be edited.

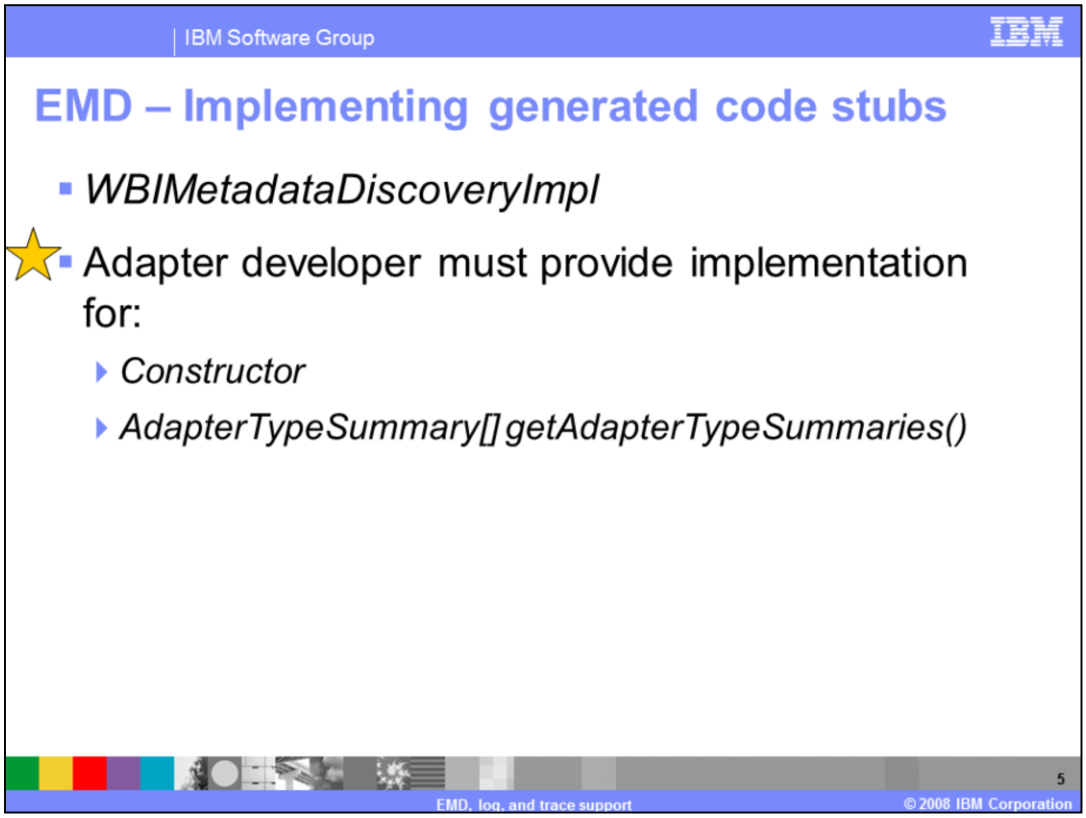
Each EMD implementation needs to have two resource bundle files. One for capturing the property group, property names and description for any properties that shall be displayed on wizard. Second one for log messages that are logged to the log file for logging and tracing.

The bundle names have to follow the convention of having EMD.properties file and LogMessages.properties in the EMD package. You need to add these files under emd package. The property names need to follow a naming convention for the bean objects. The property groups representing resource adapter, managed connection factory and ActivationSpec need to have property names matching with the bean properties. You must provide the discovery-service.xml, EMD.properties file, LogMessages.properties file and

implementation for the stubs generated.



This slide provides more details on the property groups**.** All the properties used in the discovery service are represented by the PropertyGroup set of interfaces. A property group is a collection of properties. A property group can be associated with the inbound, outbound connection configurations, MetadataTree, nodes of the MetadataTree like the MetadataObject and MetadataSelection. Property group can be a nested. It can include child property groups. Property groups provides a listener and an event interface to trickle changes from one property into another property or property group. As part of adapter foundation classes, complete implementation of these APIs is provided. You need not implement these but use the implemented API while providing implementation for EMD classes.



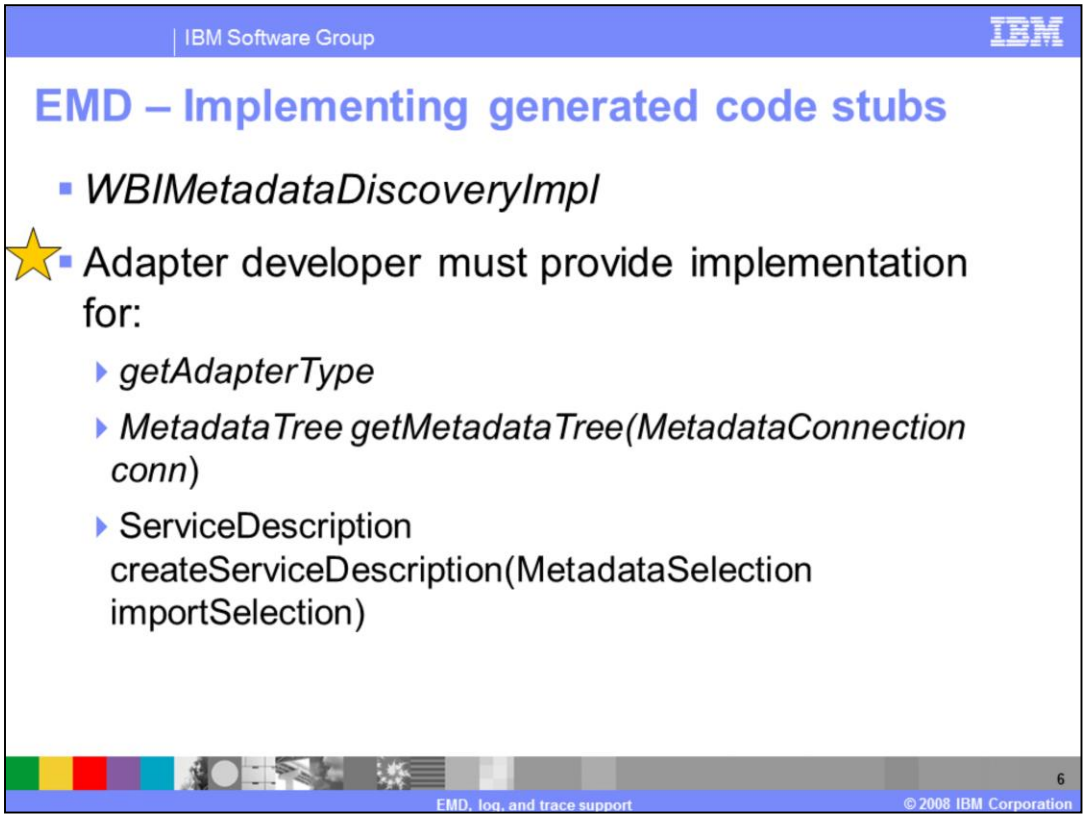
Any interaction between a tool and an EIS originates with an implementation of the **WBIMetadataDiscoveryImpl**. This is the main entry class for invocation of EMD.

You should implement the constructor and call super constructor and pass the bundle name. The bundle Name is the name of the resource bundle that has to be used for property group or properties of the EMD.

getAdapterTypeSummaries() must be implemented and should return a list of adapter type supported by EMD implementation. For jca adapters, there is only one adapter type. The adapterType must be an instance of type **WBIAdapterTypeImpl**. Each instance must be populated with information about inbound and outbound connections.

You should create instances of **WBIOutboundConnectionTypeImpl** and **WBIInboundConnectionTypeImpl** subclasses and add the connections using the addinboundconnectiontypes and addoutboundconnectiontypes methods. **WBIAdapterTypeImpl** constructor needs a set of parameters that need to be passed. The name of the class representing the ResourceAdapter class which is used to create property groups for ResourceAdapter in EMD, the number of outbound connections, and the number of inbound connections.

You need to set **supportedInMetadataService** as true for connections which can be selected in the tool to perform discovery. Add the connections to adapterType using **addInboundConnectionType** and **addOutboundConnectionType** methods



The **getAdapterType** method should return the instance of adapterType for a given ID. If the EMD implementation supports only one adapterType then it should just return that else it needs to return the one that has the input ID.

The implementation of **getMetadataTree** method should return an instance of WBIMetadataTreeImpl implementation. Each EMD implementation should extend WBIMetadataTreeImpl class and an instance of that class should be returned from this method.

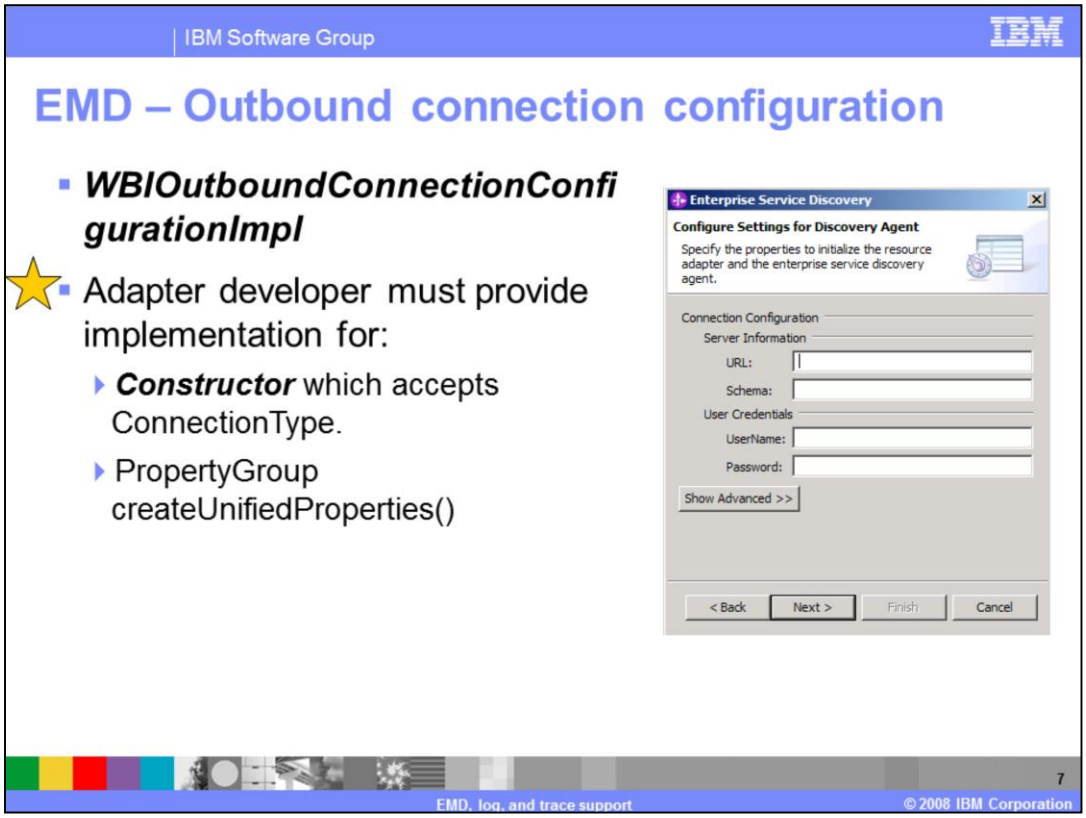
createServiceDescription method should return an instance of inbound or outbound service description depending on input selection set.

You can iterate on imported configurations (represented by MetadataSelection set) and the use the properties specified (on MetadataSelection) to complete the service description.

The instance of service description created should be filled in with name, namespace, function descriptions and configurations

After the tool gets handle to the metadatadiscoveryimpl it gets a list of adapter types. JCA adapters discovery service does not support multiple adapter types . So the tool will not

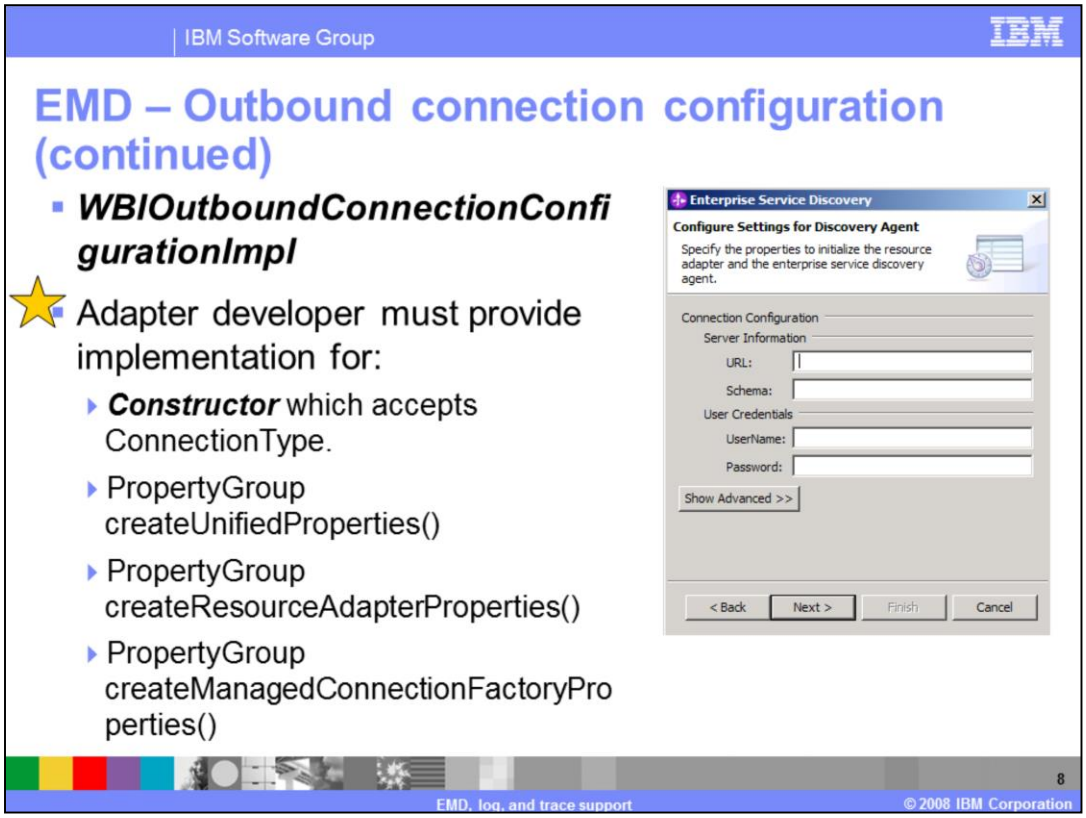
display it. This is the first selection user makes when running emd process.



For connection types that are used to perform metadata discovery **OutboundConnectionConfiguration** represents properties needed to connect to EIS for discovery and to create service descriptions. For connection types that are used only at runtime this represents properties from managed connection factory and resource adapter.

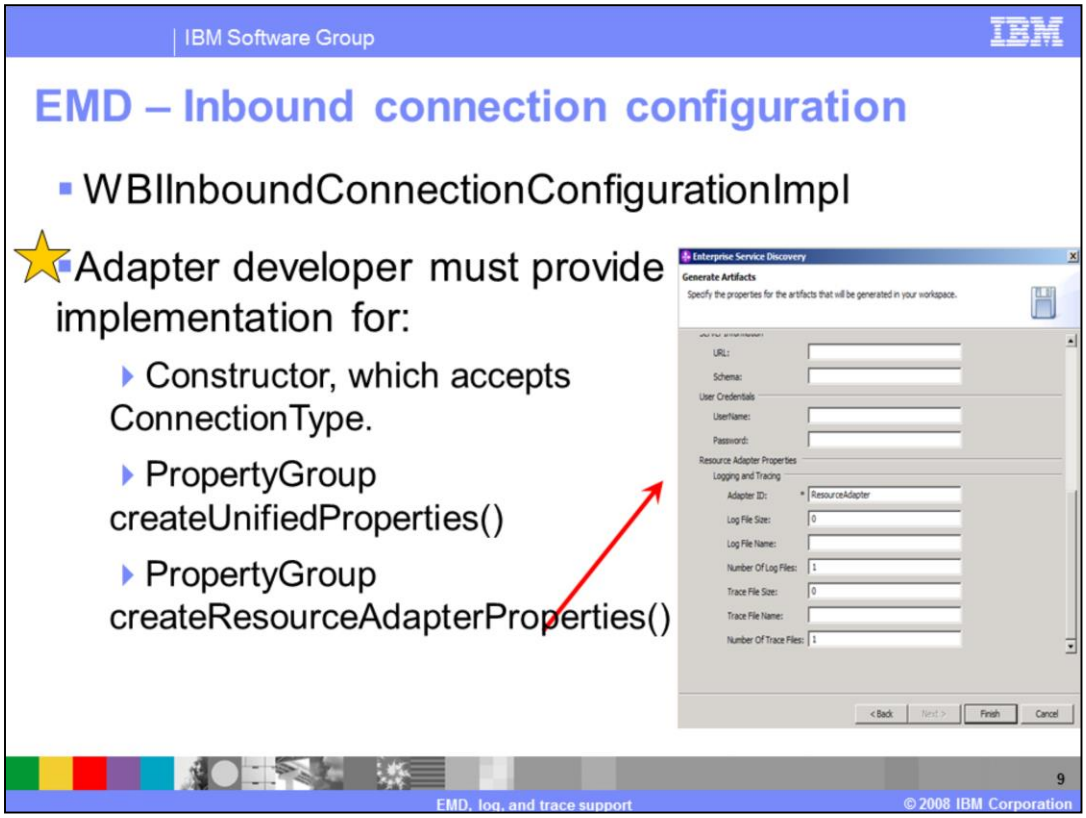
You must implement the constructor and send **ConnectionType** as parameter

createUnifiedProperties method should return an instance of property group which represents managed connection factory and resource adapter properties together in a consolidated property group. Since the property groups needed for discovery can be different then what is needed at runtime, this method can have a check on isSupportedInMetadataService and then create the corresponding property groups. For connections that are set as true for isSupportedInMetadataService, this method is used by tool to display the first connection configuration screen for EMD.



createResourceAdapterProperties method should return an instance of property group which represents properties that user can configure for resource adapter Bean. The method getPropertyGroup() in EMDUtil class can be used to fetch the properties for base class WBIResourceAdapter bean. You can add your own properties for the resource adapter bean class.

createManagedConnectionFactoryProperties method should return an instance of property group which represents properties that user can configure for managed connection factory Bean. The method getPropertyGroup() provided in EMDUtil class can be used to fetch the properties for base class WBIManagedConnectionFactory bean. You can add your own properties for there managed connection factory bean class

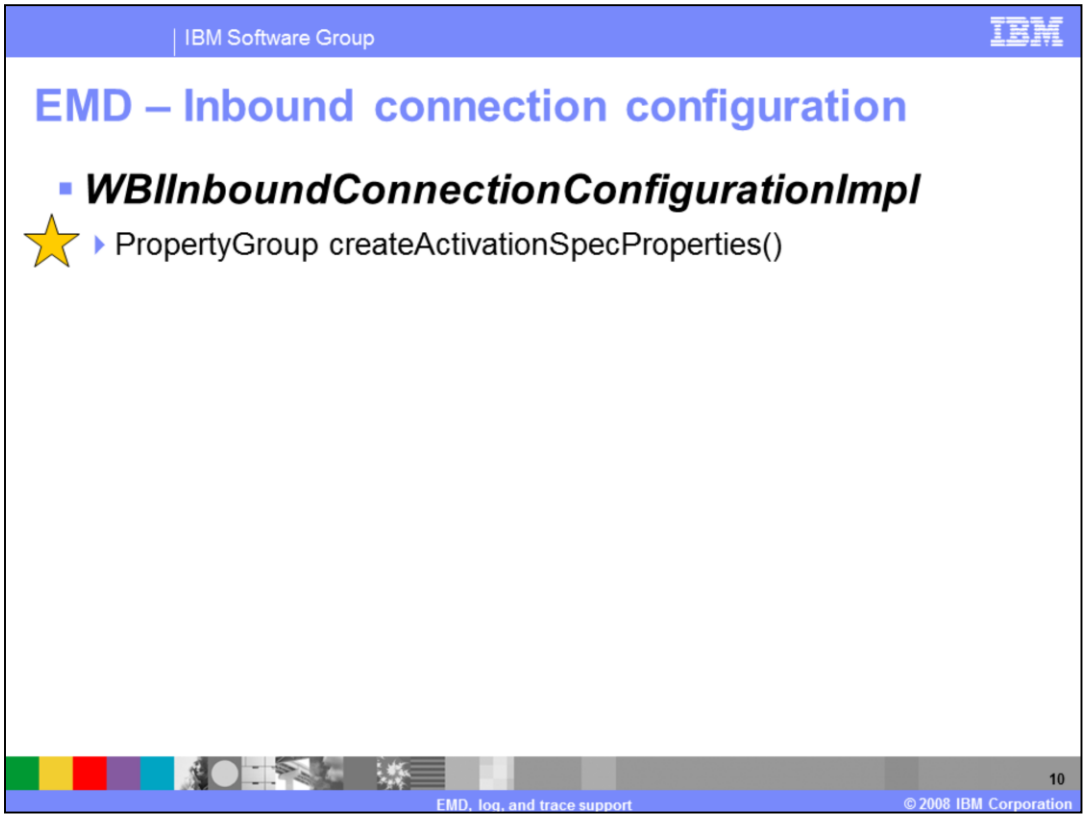


InboundConnectionConfiguration class is similar to WBIOutboundConnectionConfigurationImpl except instead of managed connection factory properties, the bean class to be handled is ActivationSpec.

You must implement the constructor and send **ConnectionType** as parameter

createUnifiedProperties method should return an instance of property group which represents activation specification and resource adapter properties together in a consolidated property group. The isSupportedInMetadataService should always be set to false

createResourceAdapterProperties method should return an instance of property group which represents properties that user can configure for resource adapter Bean. The method getPropertyGroup() in EMDUtil class can be used to fetch the properties for base class WBIResourceAdapter bean. You can add your own properties for the resource adapter bean class.



createActivationSpecProperties method should return an instance of property group which represents properties that user can configure for activation specification bean. User fills in the properties for the connection configuration in the discovery wizard. Discovery service establishes a connection to the EIS repository to perform discovery by mapping the user provided properties to managed connection factory properties and calling the getConnection() method.

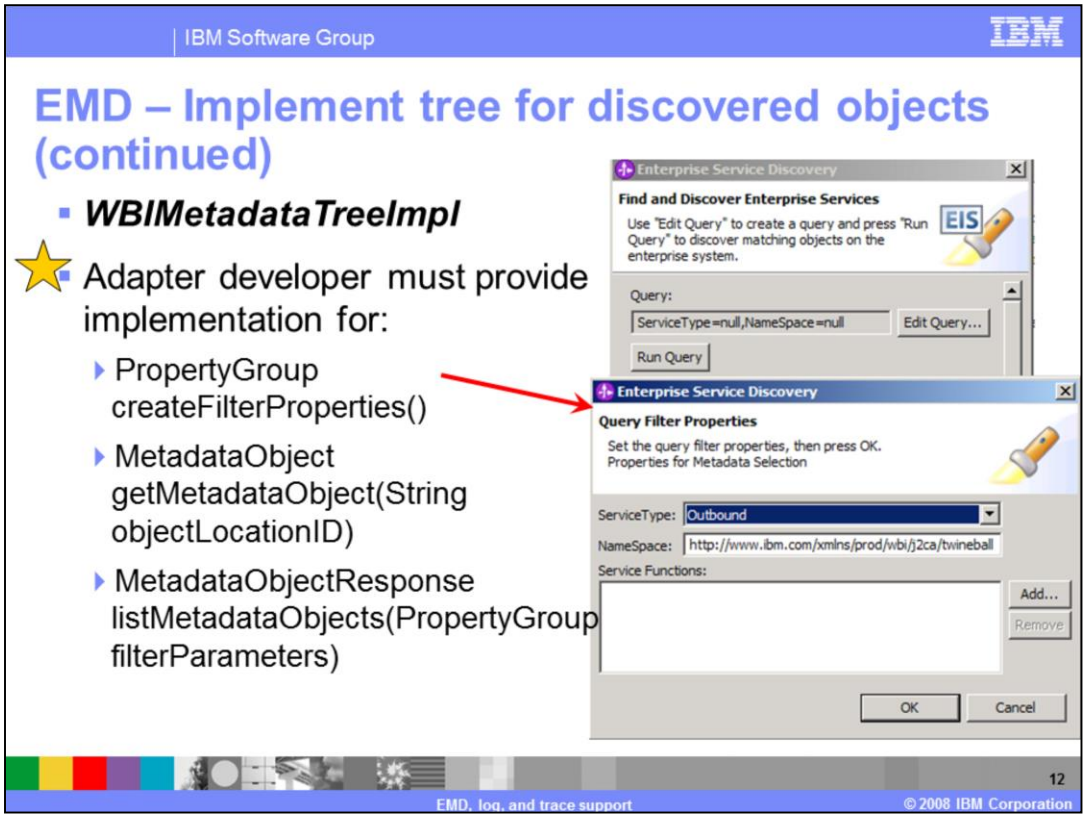
Metadataconnection object represents this connection. Interface is implemented by base classes and no implementation is needed from you. Once the connection is successful , the tool makes a request to get the metadata tree that represents object structure in EIS

WBIMetadataTreeImpl class represents the tree structure displayed in the tool that shows the object structure of data objects in EIS

The structure of the Metadata Tree is dependent on discovery service implementation. You can either choose to display the display the properties, parameters in the tree as nodes or not. recommendation is to display the leaf level properties only if there is an advantage to the user, in most cases the simple properties or parameters of the function should not be added as nodes in the tree. Node representing the object or the function should provide information about the node. For example, in EISs where function overloading is possible the function description for the node should identify the parameters in some sense so that the user can make the right selection for import.

You must provide a **constructor** that accepts **MetaDataConnection** object as parameter.

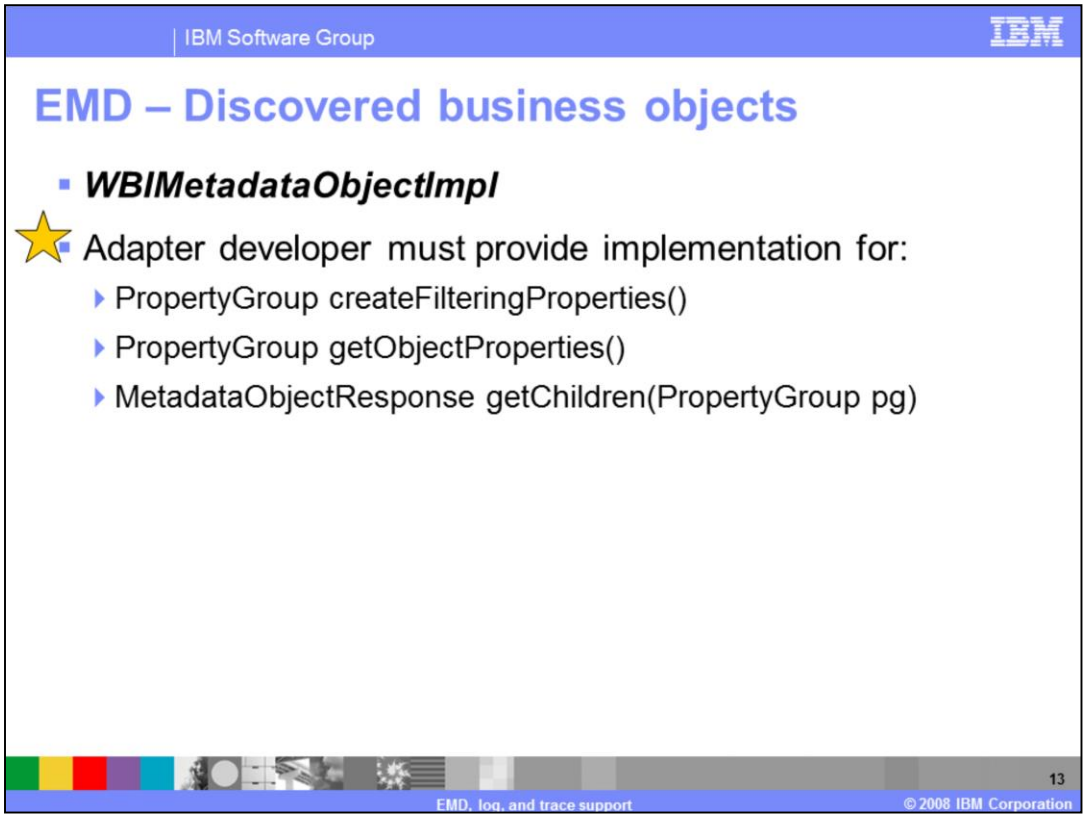
createMetaDataSelection method should return an instance of the specific MetadataSelection class. Your implementation should extend WBIMetadataSelectionImpl and return an instance of that class in this method.



createFilterProperties method should return a property group instance that is used to perform filtering for nodes of the tree. (properties like inbound or outbound to restrict objects selected in one run to be of a single type). This filter is used only for displaying top level nodes on the tree

getMetadataObject method should return an instance of MetadataObject for a specific location. Each MetadataObject instance that is added to the MetadataTree should have a unique location so that when the tool calls this method implementation it can find the exact MetadataObject and return it.

listMetadataObjects method should return an instance of WBIMetadataObjectResponseImpl. The instance should be populated with MetadataObjects using method setObjects(). Any metadataObjects that can be selected by user for import should be set as **true** with method **setIsSelectableForImport()** method.

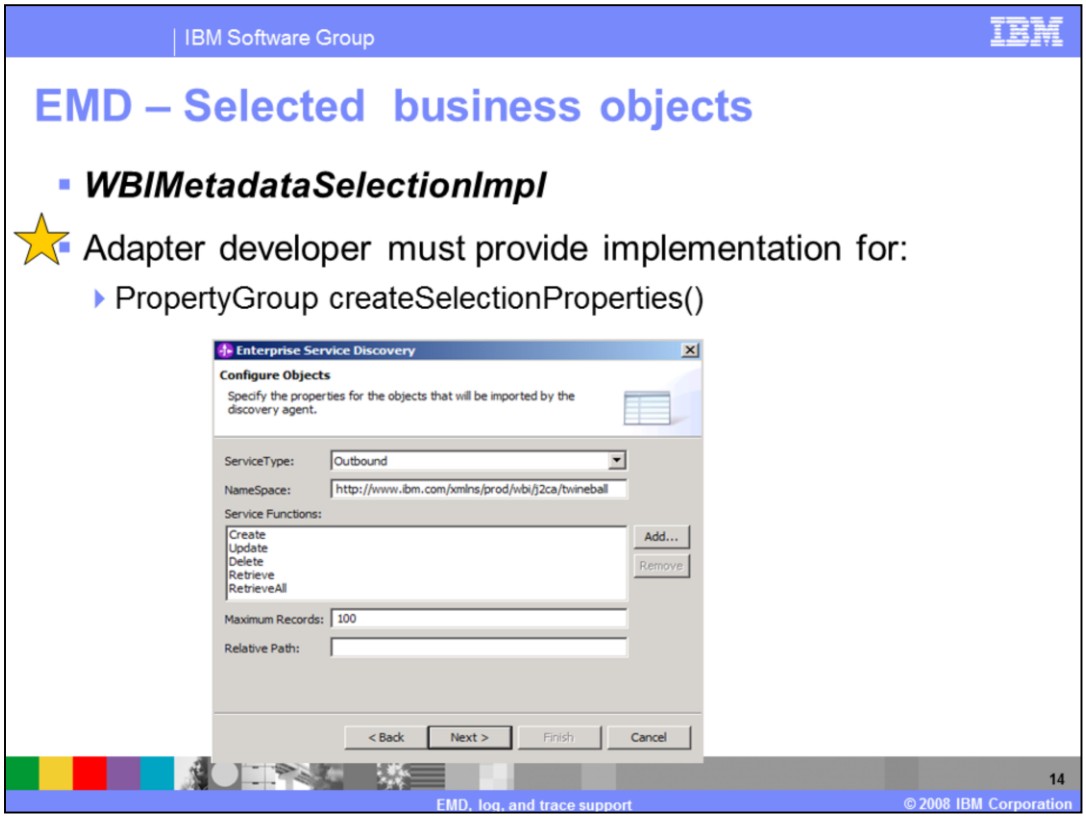


WBIMetadataObjectImpl class represents the nodes of the tree that gets displayed to the user in discovery tool. In most cases these will map to objects in EIS that the user needs to select for importing to service description

Your implementation for **createFilteringProperties** method should return an instance of a property group that represents properties that can used in filtering for fetching child objects of a MetadataObject

Your implementation for **getObjectProperties** method should return a property group that gives more information about the specific object in MetadataTree which provides information to the user about the metadata object instance. You should return null if such properties are not applicable

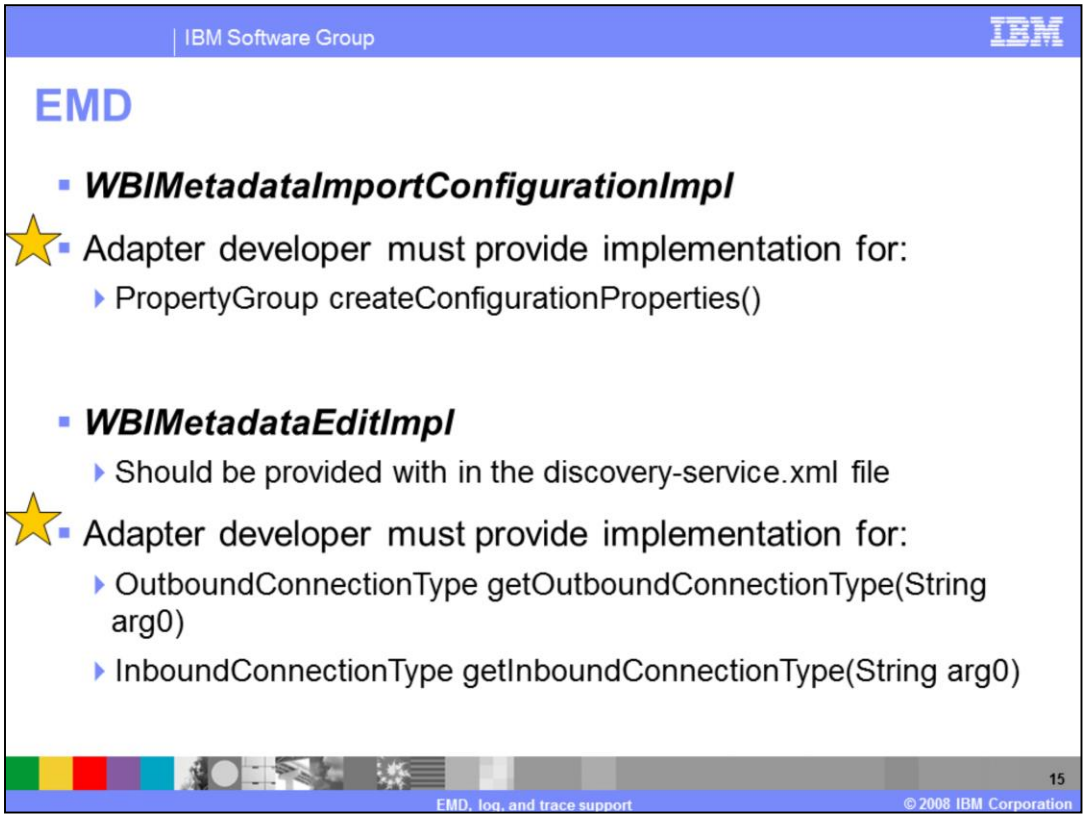
Your implementation for **getChildren** method should return an instance of WBIMetadataObjectResponseImpl populated with child MetadataObjects.The logic should use filter properties if they are supported by the implementation



WBIMetadataSelectionImpl class represents the object that holds the metadataObjects selected by user for importing. It holds properties that can be specified by the user for the whole selection set and not a specific MetadataObject.

Your implementation for the **createSelectionProperties** method should return a property group that is used to capture inputs from user needed for the whole selection. For example, specification of service type whether its “Inbound” or “Outbound”,

Namespace for business objects, operations supported, location for xsds should be saved relative to the module project.



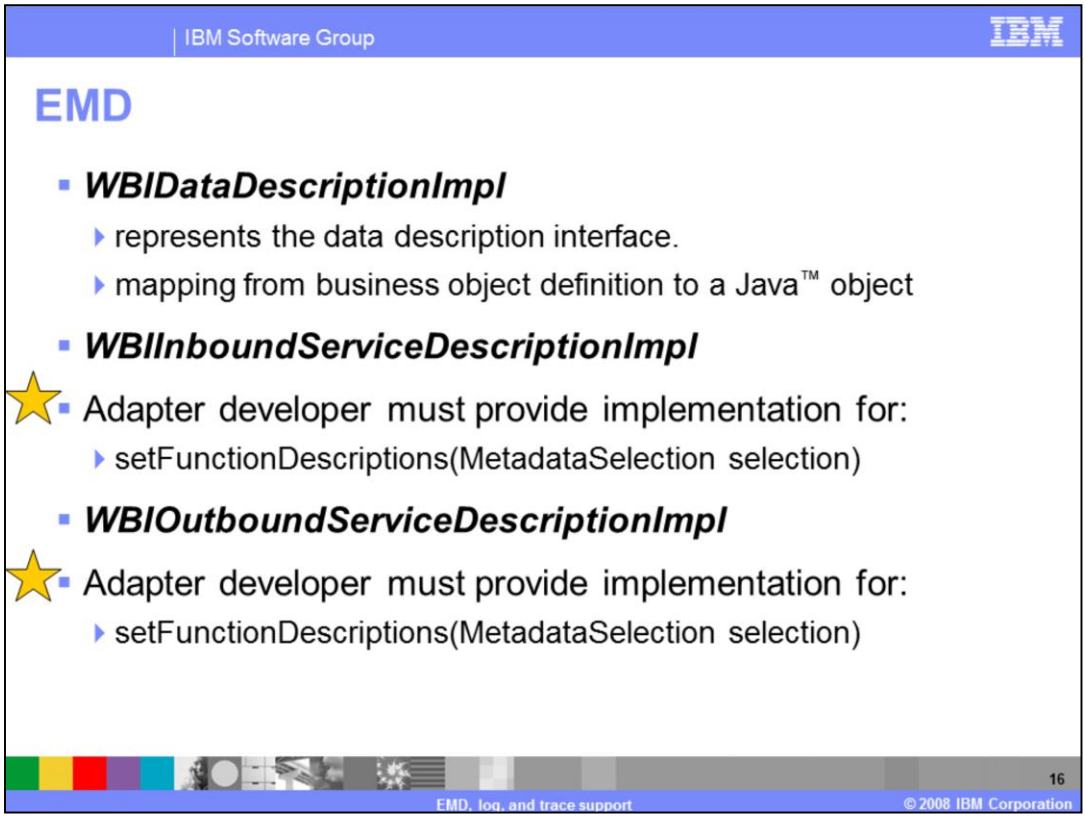
WBIMetadataImportConfigurationImpl class represents the MetadataObject selected by the user along with any configuration properties that user can specify for the specific instance of MetadataObject

Your implementation for **createConfigurationProperties** method should return a property group that is used to capture inputs from user needed for the metadata object. These are the properties specific to the instance of MetadataObject. For example, if implementation supports selection of operations for each MetadataObject or some additional information that is needed to process the object at runtime that information can be captured here.

WBIMetadataEditImpl class that is used by discovery service tool to get a handle to connectionTypes which can be used to edit properties of resource adapter, managed connection factory or activationspec. An instance of this class is created by the tool during the boot strap process. Along with the name of MetadataDiscovery class the name of this class must be specified in discovery-service.xml file.

You must provide implementation for **getOutboundConnectionType** method and return the instance of OutboundConnectionType which is used at runtime. For inbound, you must provide implementation for **getInboundConnectionType** method and return the instance

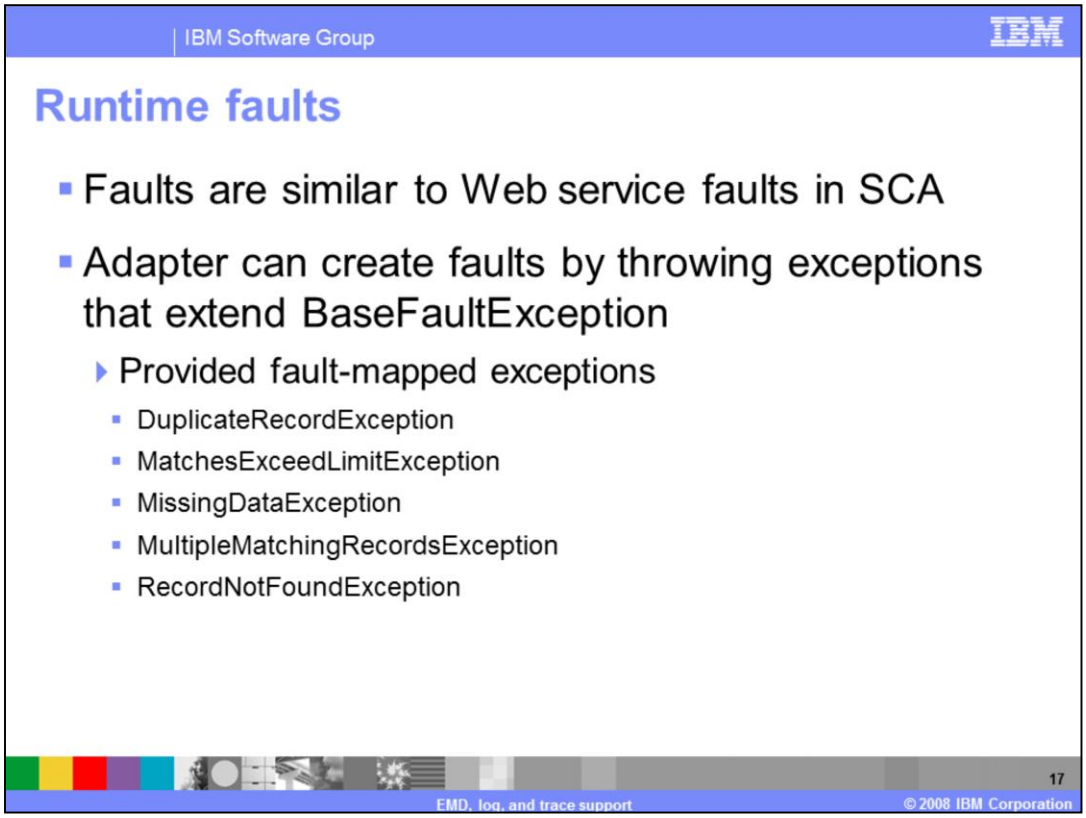
of InboundConnectionType which is used at runtime



WBIDataDescriptionImpl provides the data description that is common to the inbound and outbound models. You will implement methods in the class that include a definition of the structure and content of adapter business objects that is passed between the client and adapter at runtime. Data description enables the client to create the proper data objects for requests and interpret the data objects returned as responses. Each DataDescription instance should have a unique namespace.

You should implement **setFunctionDescriptions** method in the **WBIInboundServiceDescriptionImpl** class. **WBIInboundServiceDescriptionImpl** class represents the object that is responsible for population of function descriptions for inbound service description. **setFunctionDescriptions** method should populate the function descriptions based on objects selected in MetadataSelection and selection properties

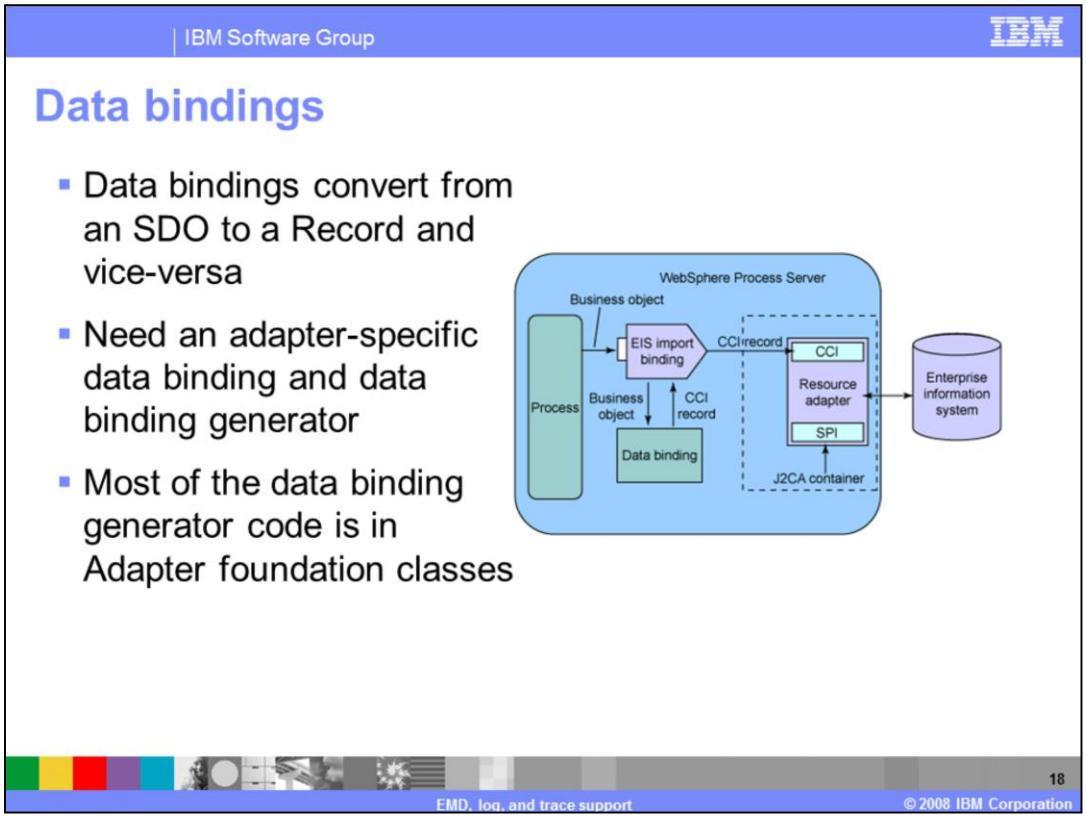
You should implement **setFunctionDescriptions** method in the **WBIOutboundServiceDescriptionImpl** class. **WBIInboundServiceDescriptionImpl** class represents the object that is responsible for population of function descriptions for outbound service description. **setFunctionDescriptions** method should populate the function descriptions based on objects selected in MetadataSelection and selection properties



A JCA EMD service can define “faults”. These faults are very similar to Web service faults from the client perspective.

JCA does not define faults, but rather it defines that connectors should throw a ResourceException when an operation cannot be performed. EMD defines a “Fault Binding” that allows an exception to be translated into a fault payload. Adapter Foundation Classes provide a standard fault binding, and some exceptions that will automatically be translated to faults.

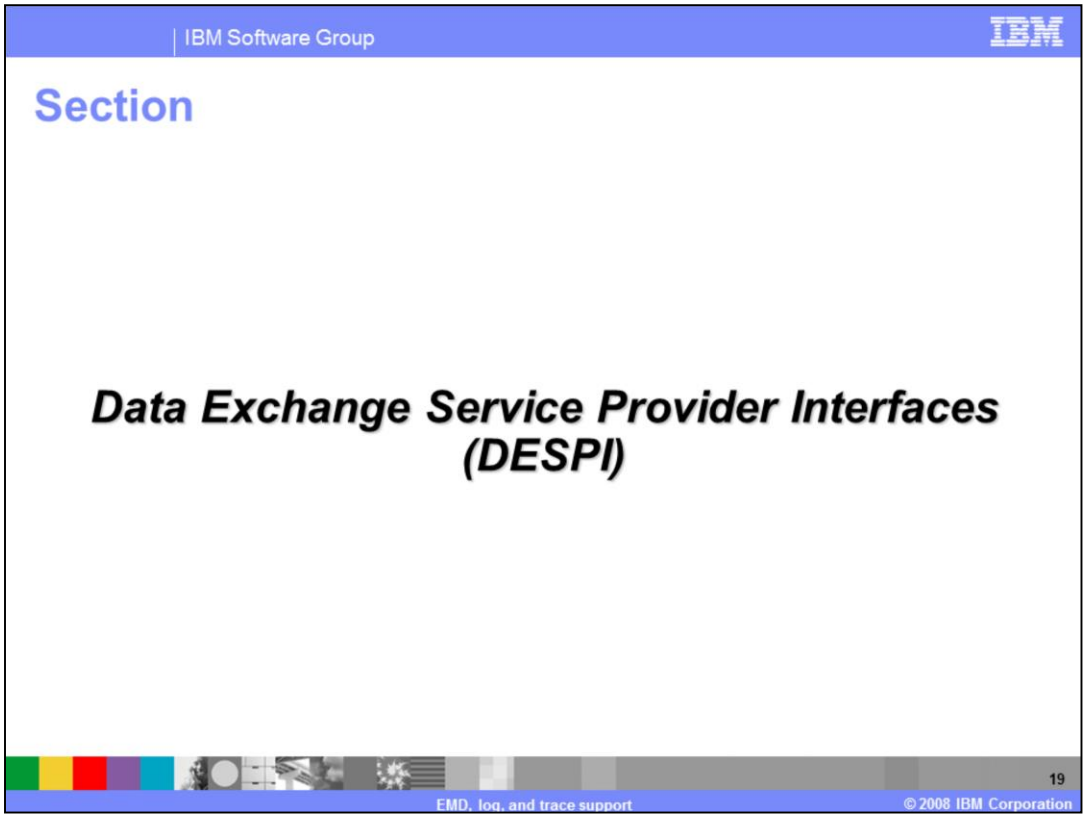
In V6.1 there is no support to configure fault generation in the discovery tool. The tool used for discovery in WebSphere Integration Developer is called external service wizard. You must add the fault information manually to the service after generation if you choose to use faults.



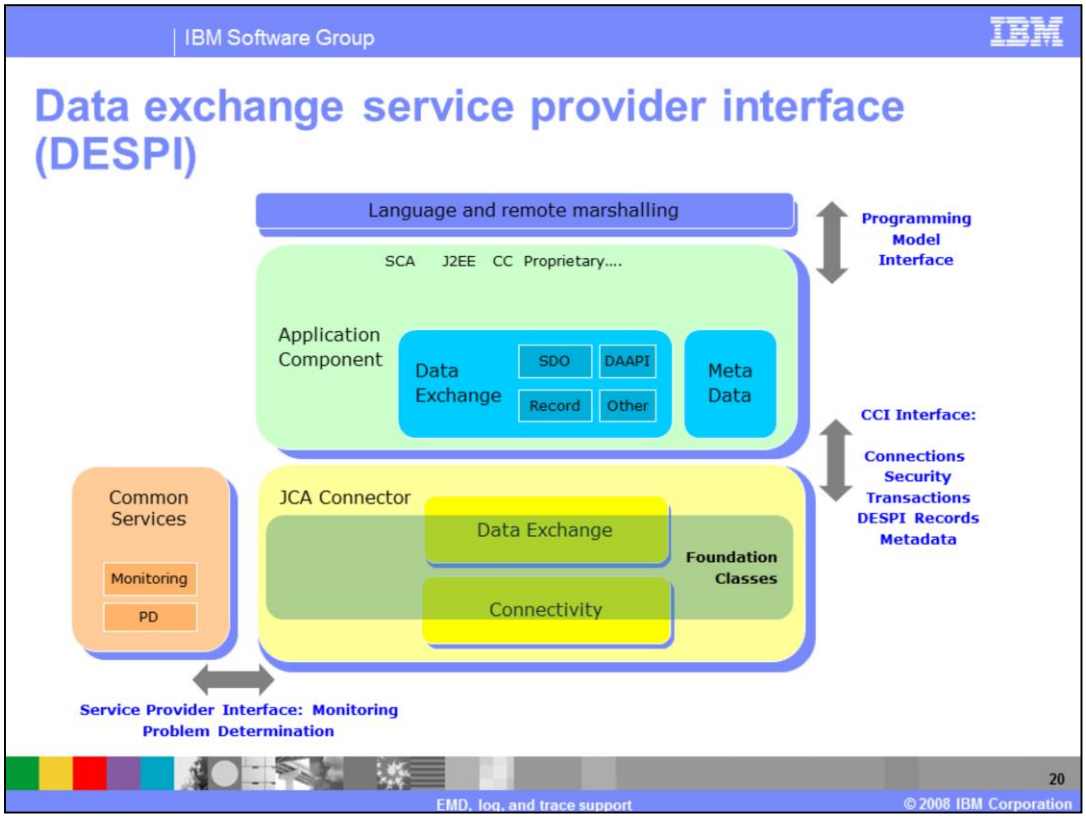
Data bindings are used to convert data format from an SDO to a common client interface record and vice versa. In version 6.0.2, WebSphere adapters used a single data binding for all EIS adapters. In V6.1, data binding “generator” is used.

When the adapter is deployed at the node level, the adapter does not have direct access to the business object definitions in an EAR file. The data binding needs the name of the object at runtime, so it can produce the correct object. This is done by including the object name into a generated data binding. The “generated” portion of the code by the WebSphere adapter toolkit is minimal, only including the correct object name.

You must provide a data binding that the data binding generator will extend. This data binding should contain all the logic necessary to transform your adapter’s record object into an SDO and vice-versa.

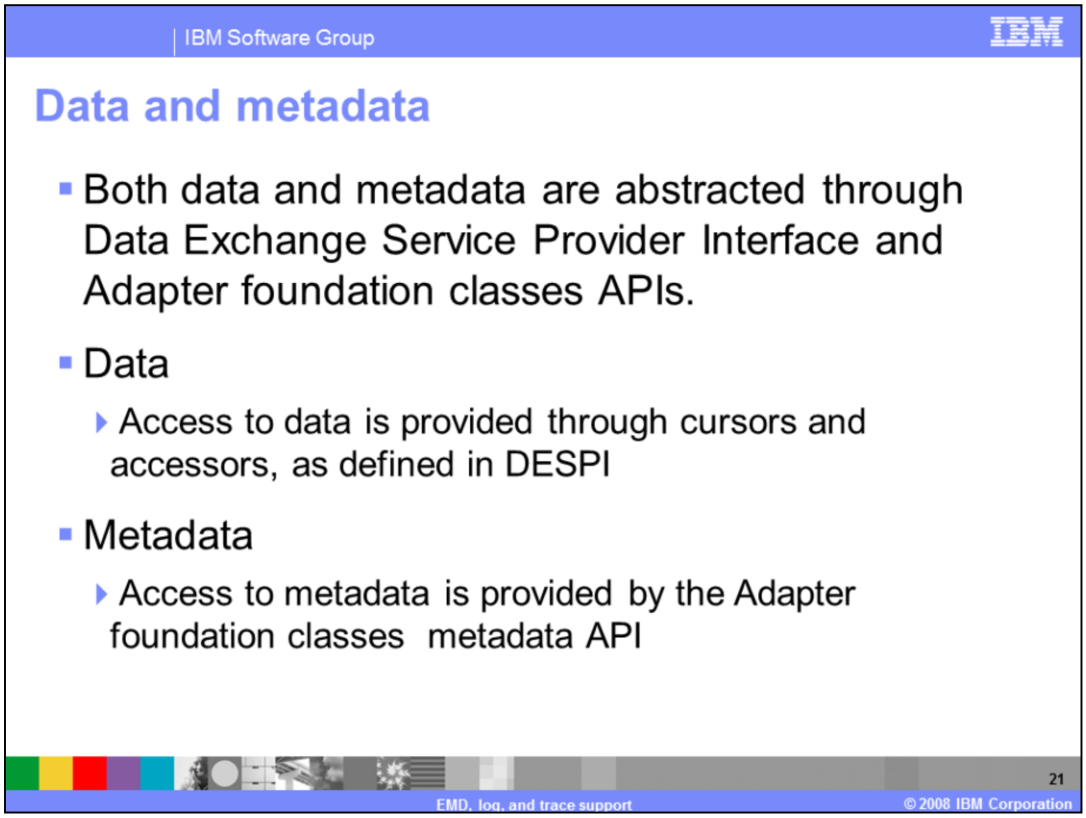


The next section provides a brief overview of the Data Exchange Service Provider Interfaces also referred to as DESPI.

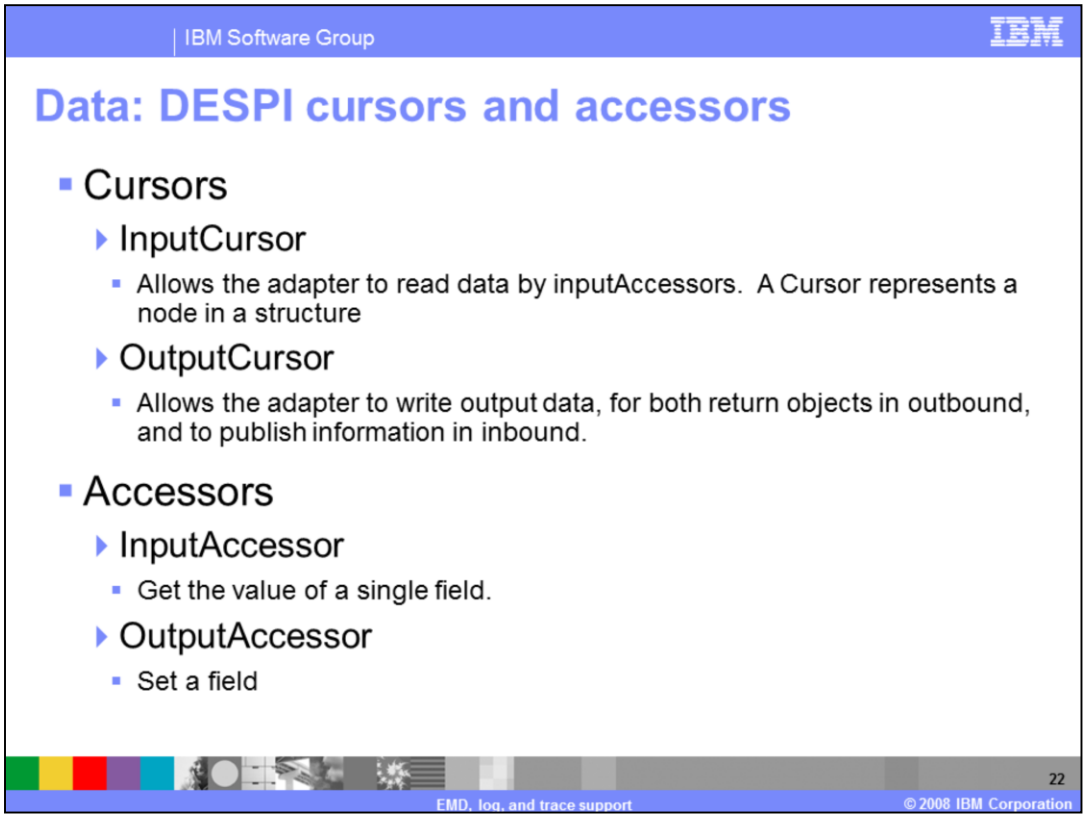


The Data Exchange Service Provider Interface, new in version 6.1, is the adapter's way of abstracting the data format, so one adapter with one code path can handle multiple data representations. This is why an IBM WebSphere Adapter can run in WebSphere Message Broker, WebSphere Transformation Extender, WebSphere Process Server, WebSphere Enterprise Service Bus, and WebSphere Application Server all using the same code.

Adapter foundation classes provide data exchange "plug-ins" that allow mediation of the SDO and Java bean data formats. WebSphere Transformation Extender and WebSphere Message Broker provide their own data exchange implementations. The adapters at the code level are unaware of which implementation is used.

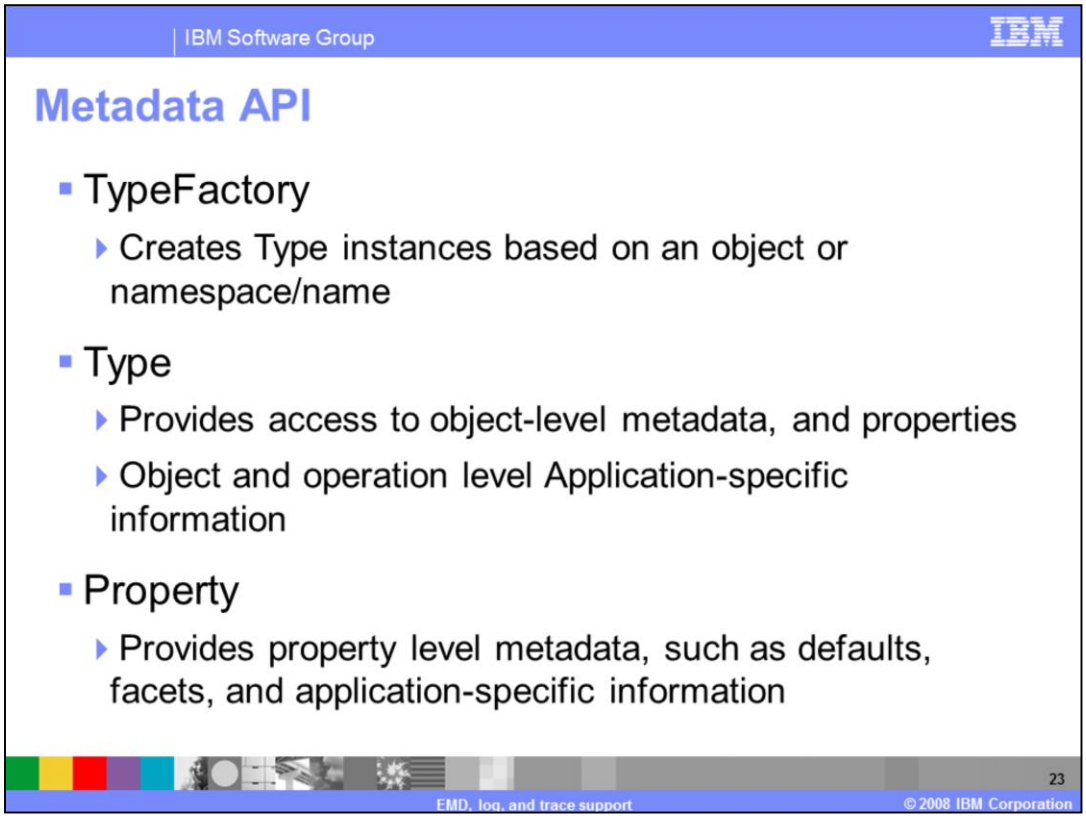


Both data and metadata are abstracted through Data Exchange Service Provider Interface (DESPI) and Adapter foundation classes APIs. Access to data is provided through cursors and accessors defined in DESPI. Access to metadata is provided using the adapter foundation class's metadata API.



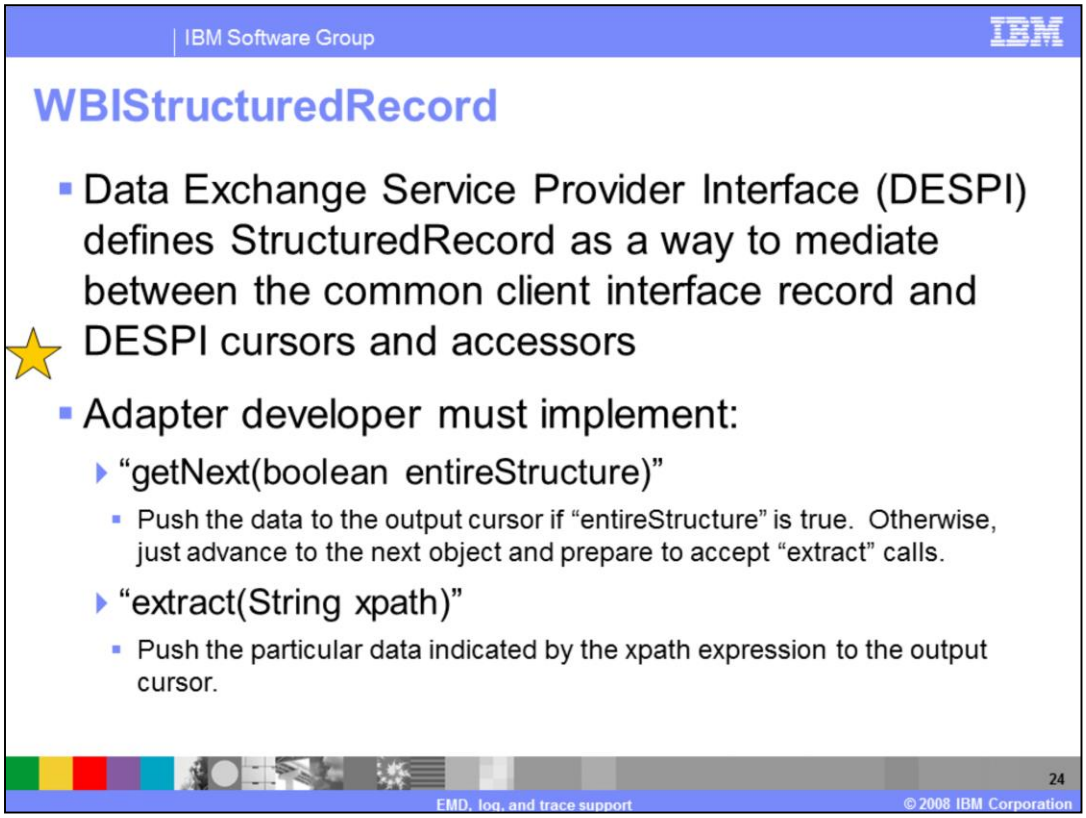
In Data Exchange Service Provider Interface (DESPI) , data is accessed with cursors and accessors. A cursor represents a node in the structure. You can use the cursor APIs to traverse the structure, get the next object in a sequence of children, and so forth. An accessor represents a singe property or attribute. The accessor API allows you to manipulate a single piece of data, either getting or setting that data.

Note that cursors and accessors are split by direction (Output and Input). The adapter should take in an InputCursor, and only read data from it, and should write data to an Output Cursor. The architecture explicitly forbids writing to an InputCursor.



The “Type” API allows the adapter to process metadata independently of the metadata format. You will notice that these classes and methods are similar to what you’d find in SDO API, and that’s intentional. The Type and Property API allow you to access to the same kinds of metadata across different formats, including parsed and cached Application Specific Information (ASI)

Note that all calls to Type, TypeFactory, and Property are cached using HashMaps. The first call with a given set of parameters can be slower, but subsequent calls should be very fast. Do not implement your own caching in your adapter-specific code. Doing so will only waste memory.

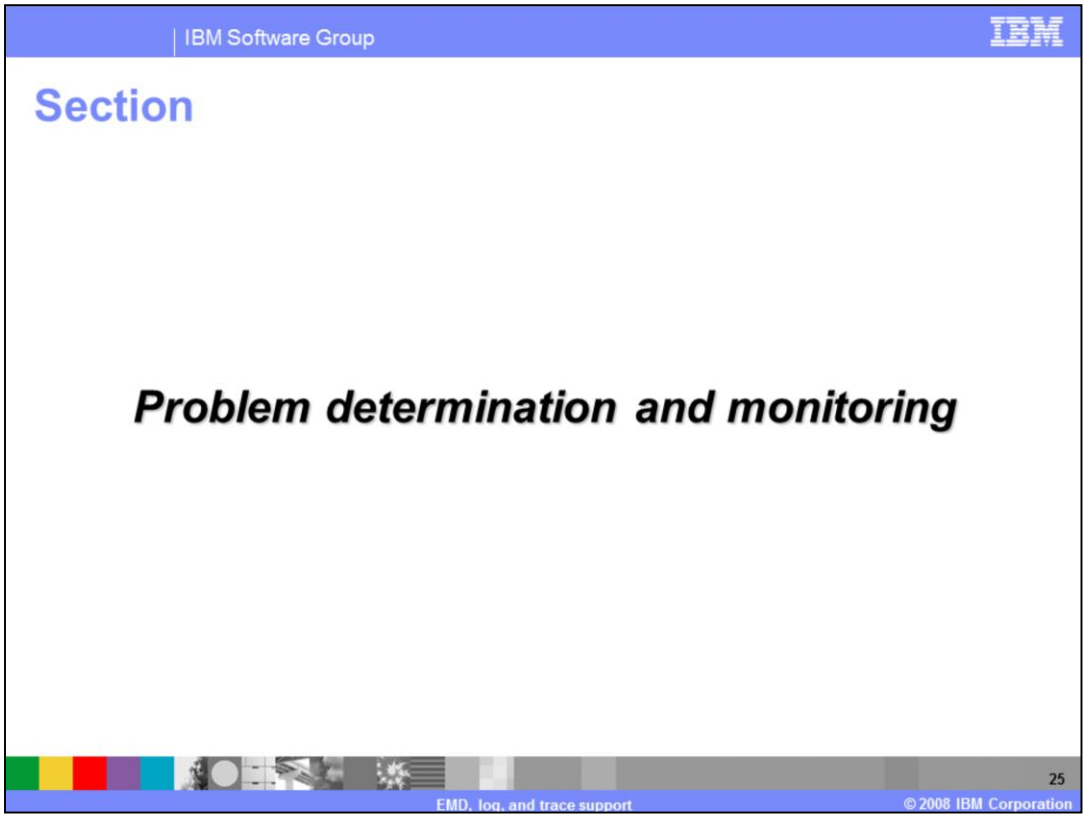


A StructuredRecord is what bridges the gap between the JCA common client interface data model (Record) and the Data Exchange Service Provider Interface.

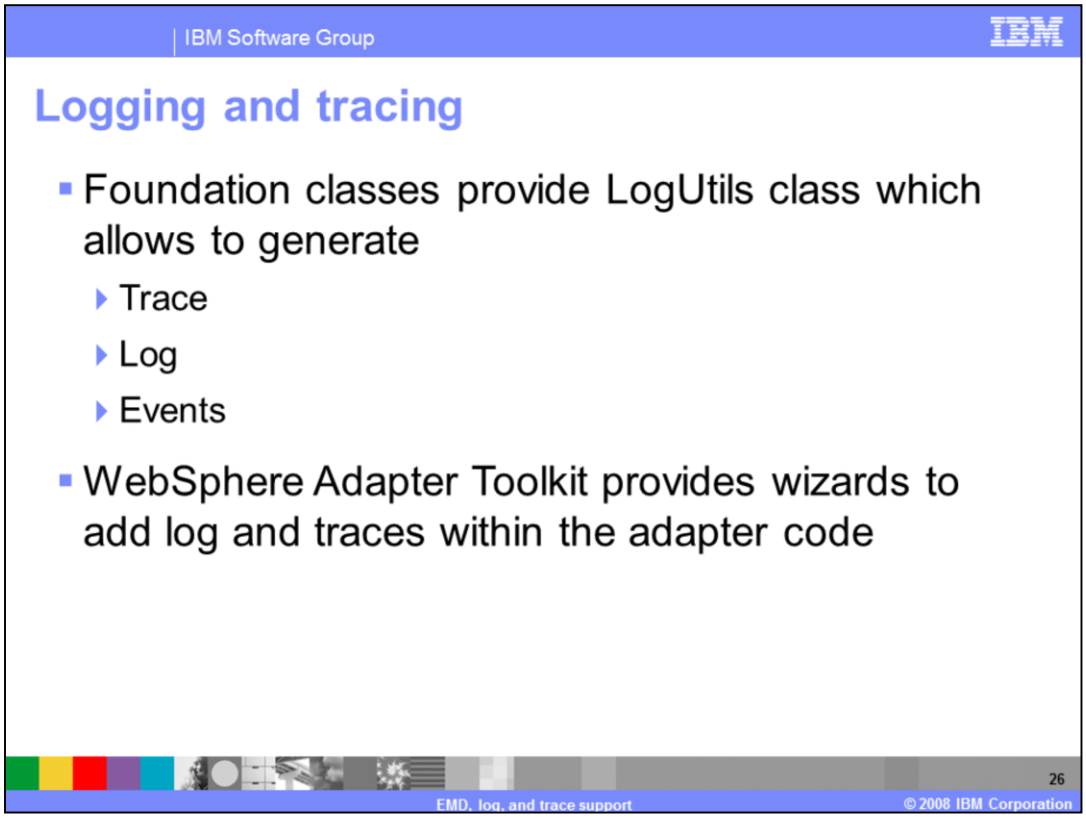
Each adapter has its own StructuredRecord interface. When processing the data in the outbound request path, the adapter will get the top-level input cursor from the input structured record and read the data from that. In the return path, or inbound, the getNext() and extract() methods are used to push the data to the output cursor.

When the adapter returns a structuredrecord from an outbound call, the record will not contain populated cursors or transformed data. The record will only contain the raw data from the EIS or a connection to the EIS and a key to the data. When the data binding calls "getNext()", the structured record implementation must advance the cursor and transform this raw data by setting the output cursors and accessors. If the parameter passed to getNext() is false, the structured record should only advance the cursor and not transform the data. This is designed to be used in conjunction with the extract() call, which will set a particular output accessor.

In summary, if the parameter passed to getNext() is true the method will transform the entire data into the output cursor. The combination of getNext() method with parameter set to false and extract() will allow the client to select a particular field or fields to transform.

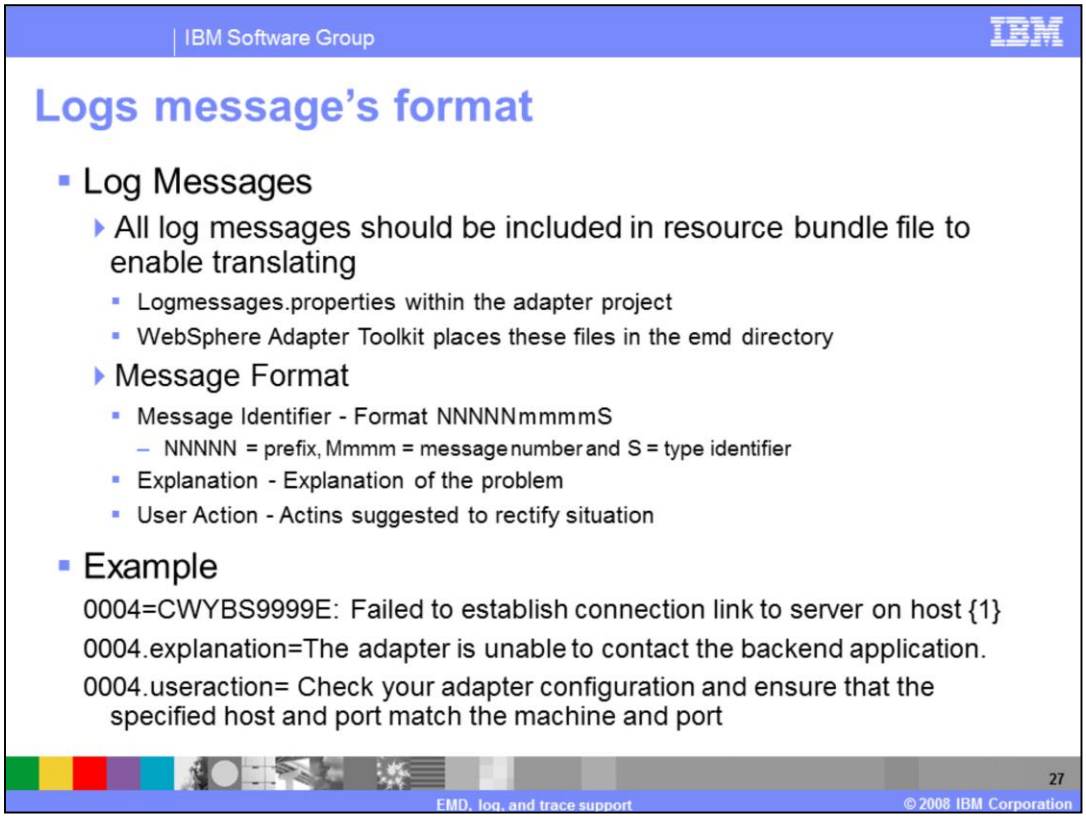


The next section provides a brief overview of the support provided by adapter foundation classes and the code stubs generated by the toolkit to support problem determination and monitoring.



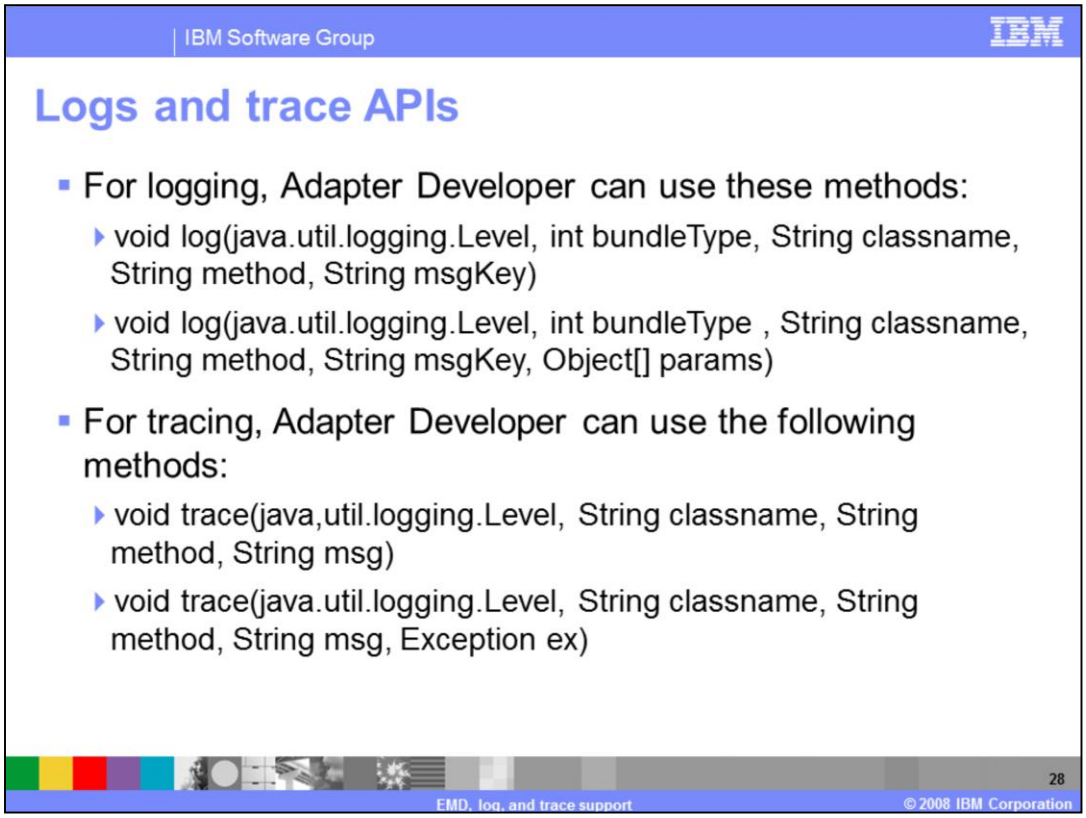
The JCA 1.5 specification provides very minimal support for communicating information to the user. It defines a single stream to which the adapter writes any information. Without additional tools or support, users of your adapter cannot filter information or analyze information.

LogUtils class enables you to target different information to different user roles and filter information. Providing information about the runtime state of the adapter is a critical aspect of adapter development. This information is invaluable to support teams trying to resolve problems and the adapter users looking to track operations and monitor the adapter.



Logs are used to convey customers of the adapter the information that is necessary to their understanding of the adapter, warnings, potential problems or errors that have occurred and any actions suggested to resolve them. The various levels of lags are fatal, severe, warning, audit and info. Log messages should not be hard coded in the adapter to enable translation. Log messages should be placed in Logmessages.properties file and should follow a defined structure as shown in the slide.

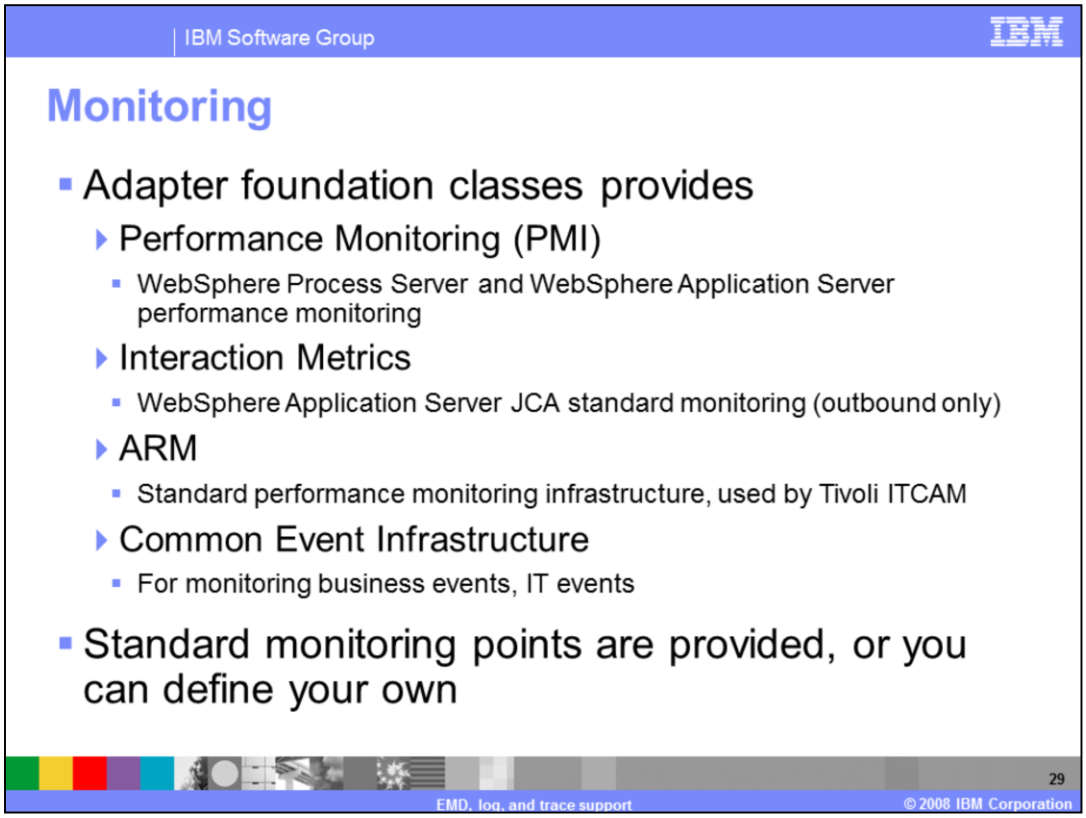
Trace information is used to convey information that is intended for support teams and developers. Information includes stack dumps for any exceptions encountered and operation logic for debugging purposes. Messages are directed at the teams that wrote or support the adapter rather than any customers, trace messages do not need to be translated and can be hard-coded in the adapter. The levels are fine, finer, and finest.



The log method of the LogUtils class should be used to generate a log message. You should provide a key to the log message you want to use from the log message file. This log method can also take optional parameters if there are values to be substituted in the message.

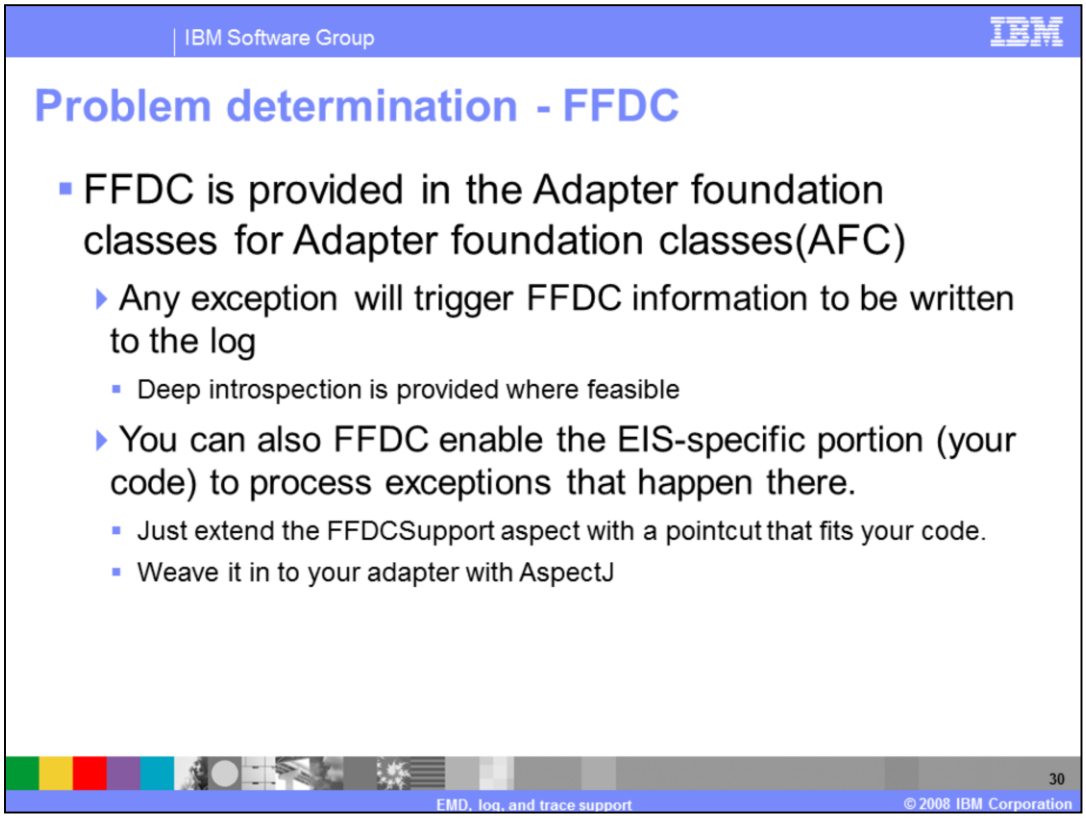
For tracing, you should check if trace is enabled before generating any trace messages because of performance issues.

LogUtils.isTraceEnabled method can be used for checking if trace is enabled.



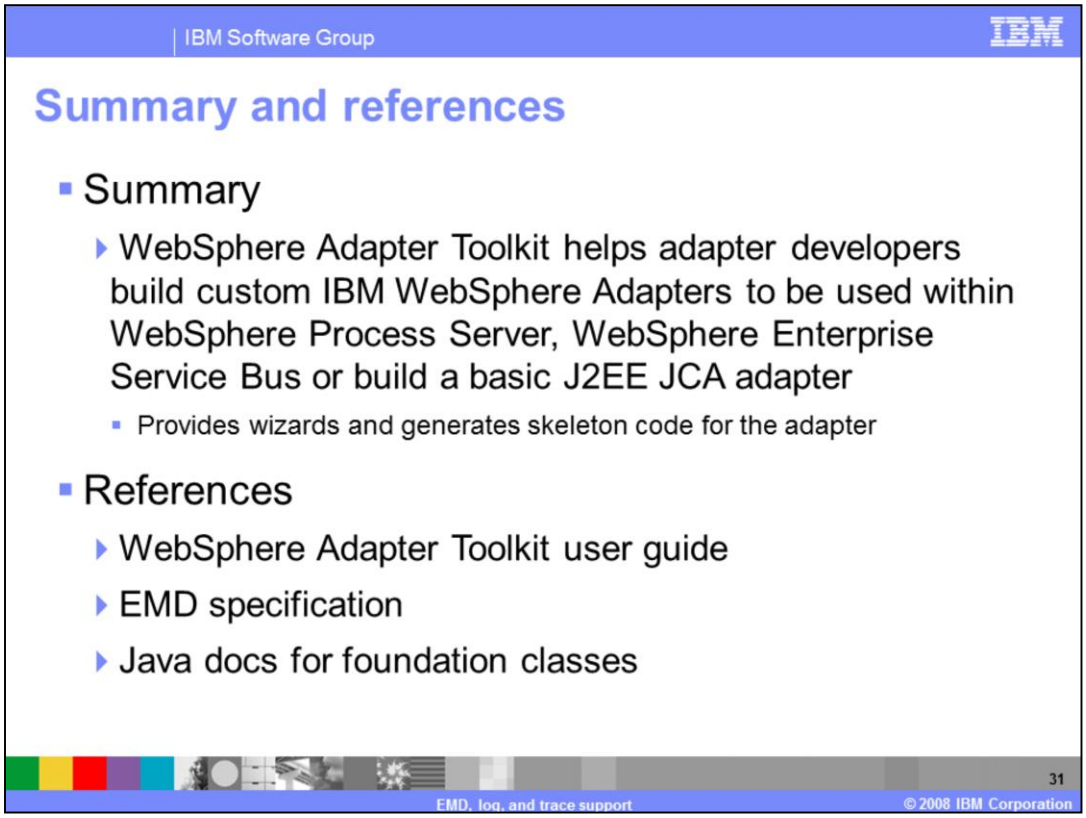
Adapter foundation classes will automatically provide a certain amount of monitoring in event delivery with polling. If you are using polling, and do not want to have any finer grained monitoring than what is provided, you need not implement more.

On the other hand, if you are implementing “callback” and want to define additional monitoring points, the APIs provided by foundation classes can be used.



FFDC stands for First Failure Data Capture. It is a mechanism to feed back exception and state information back to support and development. Normally, exceptions, stack traces, and the object where the exception occurred and its public variables are included; however it can be extended to provide more information.

FFDC is implemented by adapter foundation classes , but only for adapter foundation classes code. If you want to have your adapter-specific code enabled for FFDC, you will need to use the AspectJ compiler and “weave” in the FFDC aspect into your code.



WebSphere Adapter Toolkit helps adapter developers build custom IBM WebSphere Adapters to be used within WebSphere Process Server,WebSphere Enterprise Service Bus or build a basic J2EE JCA adapter. The underlying adapter foundation classes used by the toolkit simplify the process of adapter development by providing implementation for most generic contracts so you only provide the implementation logic for your backend EIS.

Trademarks, copyrights, and disclaimers

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

IBM WebSphere

A current list of other IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

J2EE, and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Product data has been reviewed for accuracy as of the date of initial publication. Product data is subject to change without notice. This document could include technical inaccuracies or typographical errors. IBM may make improvements or changes in the products or programs described herein at any time without notice. Any statements regarding IBM's future direction and intent are subject to change or withdrawal without notice, and represent goals and objectives only. References in this document to IBM products, programs, or services does not imply that IBM intends to make such products, programs or services available in all countries in which IBM operates or does business. Any reference to an IBM Program Product in this document is not intended to state or imply that only that program product may be used. Any functionally equivalent program, that does not infringe IBM's intellectual property rights, may be used instead.

Information is provided "AS IS" without warranty of any kind. THE INFORMATION PROVIDED IN THIS DOCUMENT IS DISTRIBUTED "AS IS" WITHOUT ANY WARRANTY, EITHER EXPRESS OR IMPLIED. IBM EXPRESSLY DISCLAIMS ANY WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NONINFRINGEMENT. IBM shall have no responsibility to update this information. IBM products are warranted, if at all, according to the terms and conditions of the agreements (for example, IBM Customer Agreement, Statement of Limited Warranty, International Program License Agreement, etc.) under which they are provided. Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products in connection with this publication and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products.

IBM makes no representations or warranties, express or implied, regarding non-IBM products and services.

The provision of the information contained herein is not intended to, and does not, grant any right or license under any IBM patents or copyrights. Inquiries regarding patent or copyright licenses should be made, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

Performance is based on measurements and projections using standard IBM benchmarks in a controlled environment. All customer examples described are presented as illustrations of how those customers have used IBM products and the results they may have achieved. The actual throughput or performance that any user will experience will vary depending upon considerations such as the amount of multiprogramming in the user's job stream, the I/O configuration, the storage configuration, and the workload processed. Therefore, no assurance can be given that an individual user will achieve throughput or performance improvements equivalent to the ratios stated here.