

EJB Assurance Services

AS DISCUSSED IN CHAPTER 17, “Assurance and Security Services,” when building enterprise systems, one must consider an infrastructure and approach not only for enabling rapid development but also imbued with systems assurance for scalability, reliability, maintainability, availability, and security. One of the primary reasons often espoused for using Enterprise JavaBeans (EJB) technology is the inherent and transparent benefits it provides for enhancing systems assurance. Despite the significance often ascribed to EJB assurance features, the consideration and proper utilization of such features is often overlooked or put on the back burner in favor of taking advantage of EJB’s rapid development benefits and layering in new functionality. This chapter focuses on describing the specific EJB assurance features and how to leverage their use in constructing enterprise systems. Specifically, this chapter discusses EJB transactions, security, clustering, and other EJB features that can be used to infuse scalability, reliability, maintainability, availability, and security into enterprise systems built with EJB.

In this chapter, you will learn:

- How to define and utilize transactions with EJB.
- How to utilize J2EE EJB container features for building scalable and available EJB applications.
- How to utilize EJB security features for building secure EJB applications.
- How to utilize JAAS for building secure J2EE and EJB applications.

EJB Transactions

We explored a wide range of Java-related transaction management concepts and services in Chapter 14, “Transaction Services with JTA and JTS.” Transactions are critical to ensure the reliability of your enterprise applications. Without atomicity, consistency, isolation, and durability (ACID) principles embodied by transactions, the reliability of your applications is tremendously affected.

Transaction services for EJB components are implemented within the confines of the EJB container/server environment. EJBs can use such services either declaratively or programmatically. Declarative transaction management (aka container-managed transaction demarcation) involves the specification of transaction attributes associated with EJBs via deployment descriptor parameters. Programmatic transaction management (aka bean-managed transaction demarcation) involves using JTA APIs to manage transactions associated with EJBs. This section describes the means for using transactions with EJBs according to both models of transaction demarcation, as well as general considerations for providing transactions for EJBs.

General EJB Transaction Considerations

A few general facts must be considered and absorbed before we describe how to use transactions with EJBs. For one, the basic model employed by transactions with EJBs is the flat transaction model as described in Chapter 14. Nested transactions are not specifically required to be supported by vendors but may be incorporated into a future EJB specification. In addition to the basic transaction model, a basic set of APIs must be supported. The JTA is required to be supported, but the JTS for interoperability between different application servers is not required to be supported. The J2EE connector transaction-related APIs, discussed in the next chapter, however, are required.

In general, when an EJB client invokes the services of an EJB via one of its interfaces, the EJB container intercepts such an invocation before it delegates the call to an appropriate EJB instance. The container first assesses what method call is being made on what bean. The container then assesses what transaction attributes apply to that method to determine the transactional behavior of the method. Depending on the transaction attribute, the container may suspend any transaction context passed from the client, propagate any transaction context passed from the client, create a new transaction context, throw any exceptions associated with improper transaction context propagation, or neither propagate nor define any transaction context.

The container will make sure transaction context is propagated appropriately to other beans within the same container environment as needed. The container may also propagate the context to other EJB containers or perhaps to other distributed application servers via JTS. The container will also interact with a resource manager and transaction manager to coordinate and synchronize with such resources to ensure that distributed transaction processing can occur effectively. Chapter 2, “Enterprise Data,” discussed such transaction management as it applied to databases as resource managers, and Chapter 14, discussed such transaction management as it applies in general.

EJB Transaction Attributes

Chapter 14 also presented the core types of transaction attributes and their meaning. Figure 28.1 depicts the types of transaction attributes and how their specification affects the propagation of transaction context to an EJB instance. The transaction attribute types are listed, followed by an indication as to whether a transaction associated with that

transaction attribute type must ultimately be associated with the bean instance. The three columns following such information in Figure 28.1 depict from what source the transaction context may be propagated (that is, client or container), as well as whether and how it propagates to the EJB instance. An arrow with a circle at its beginning indicates that the transaction must propagate from that source, whereas an arrow without such a circle indicates that it may optionally propagate from that source.

Transaction Attribute	Transaction?	Client Transaction	Container Transaction	EJB Instance	EJB Types
Required	YES	if ----->	else ----->	----->	S Ss Sw E M
RequiresNew	YES	-----> TS	●----->	----->	S Ss Sw E T
Mandatory	YES	●----->	----->	----->	S Ss E
Supports	Optional	----->	----->	----->	S Sw Eo
NotSupported	NO	-----> TS	----->	----->	S Sw Eo M T
Never	NO	-----> TE	----->	----->	S Sw Eo

KEY		S - session bean	E - entity bean
----->	Source optionally creates transaction	TS - Transaction suspended	Eo - Entity Bean (optional)
●----->	Source must create transaction	TE - Transaction error	M - message driven bean
		Sw - session Web service	T - ejb Timeout

Figure 28.1 EJB transaction attributes.

If the container stops a client’s transaction from propagating, a “TS” indicates that it suspends the transaction and resumes that transaction after the bean instance method invocation returns. A “TE” indicates that the transaction is halted and an error condition is returned to the client. In such a case, the container will throw a `RemoteException` if the client is a remote EJB client or an `EJBException` if the client is a local client. If a transaction attribute is `Mandatory` and a transaction has not propagated from the client,

then the container will throw a `TransactionRequiredException` to a remote client and a `TransactionRequiredLocalException` to a local client. Finally, Figure 28.1 also defines those EJB types (with identifiers mapping to the EJB types indicated in the figure's key) to which each transaction attribute may apply.

EJB Concurrency

As can be inferred from Figure 28.1, not all EJB types have the same transactional behavior ascribed to them. In fact, different bean types dictate how the container will process transactions for that bean, as well as how the container processes concurrent requests to such beans. This section describes these considerations in more detail.

Table 28.1 provides a mapping for the different concurrency scenarios that may be associated with an EJB and how transactions affect the behavior of the container in dealing with these scenarios. The primary two column groupings indicate what view is being considered for the concurrency scenario—that is, whether or not the view is from the EJB client or container's perspective.

Table 28.1 **Concurrent EJB Access**

Bean Type	Concurrent Calls by Client View	Concurrent Calls at Server View	
		Same Tx	Different Tx
Stateless Session Beans	Incorrect to do	Impossible to happen	
Stateful Session Beans	Incorrect to do	Throw exception or Serialize access	
Entity Beans	Allowed to do	Non-reentrant-throw exception or Reentrant- program around	Create multiple instances or Serialize access
Message Driven Beans	Impossible to do	Impossible to happen	

Session Bean Concurrency

From an EJB client's perspective, if more than one client thread concurrently accesses a particular session bean's object interface, this is to be viewed as an application error. If the session bean is a stateless session bean, the client should simply use the bean's home interface to obtain a handle to a new object interface. If the session bean is a stateful session bean, this is generally regarded as an application error because the conversational state associated with the client by the server is supposed to be with respect to one client session.

From the container's perspective, the story is a little different. Here we have a situation in which a container receives a client request and must determine how to select and deliver that request to an EJB instance capable of handling the request. In the case of stateless session beans, a container will simply select an EJB from a pool of beans for that bean type and delegate the client request to that instance for handling. There is no

concurrency issue because the container has complete control over which bean instance will service the request.

However, consider the case of a concurrent request on a stateful session bean instance from the server's perspective. Suppose first that a request comes in for a particular stateful session bean instance and the container will handle this by delegating the invocation to the appropriate bean instance as usual. Now suppose that another concurrent request comes in for the same bean instance before the existing method invocation has had a chance to service the request. In such a case, the container should generally throw a `RemoteException` if the client is a remote EJB client or an `EJBException` if the client is a local client. In some circumstances, the container may alternatively elect to handle both requests by serializing the request from the second client and allowing it to execute after the existing method invocation returns. This can lead to unpredictable behavior, however, and clients should not rely on this being offered by your EJB container.

Message-Driven Bean Concurrency

For message-driven beans, it is impossible for clients to reference the same bean's client view because the clients are message consumers that simply produce messages to a particular topic or queue. The messaging service and EJB container deal with the delivery of such messages. In the case of message-driven beans, from the container's point of view it will simply select a message-driven bean from a pool of beans to deliver the message to for handling. There is also no concurrency issue for message-driven beans because the container has complete control over which bean instance will service the request as it did for stateless session beans.

Entity Bean Concurrency with Different Transaction Contexts

For entity beans, it is actually quite natural for EJB clients to try to talk to the same entity bean view. For example, more than one EJB client may be attempting to access and update a particular user account entity bean. The EJB clients, however, obtain handles to such beans most often via the home interface for such beans.

For entity beans, the concurrency story is somewhat different from the container's perspective. If a concurrent request comes in for a particular entity bean instance and the requests are associated with different transaction contexts, the container generally is left with determining how to handle the situation. A container may serialize access to the bean instance or it may create multiple bean instances, one for each request. In the case of serialized access to the instance, the container generally will obtain an exclusive lock on the associated data in the database. With such an exclusive lock on the associated database data, the instance does not need to have its state synchronized with the database every time a new transaction begins and is associated with the instance.

In the case in which multiple bean instances are created to handle requests, the resource manager (that is, database) is left with the chore of synchronizing the concurrent access when such calls result in database access calls. The container may either return such instances to the pool or cache them in some way when the instances are finished

with their transactions. In either case, when another transaction is begun and associated with such instances, the container must invoke `ejbLoad()` on the instances to load their state and ensure that they are in sync with the database. When such instances are being updated inside of a transaction, the container might acquire a temporary exclusive lock on the database while the instance's method under the transaction is being processed. This, of course, could lead to a deadlock situation if concurrent locks are sought by different transactions. Alternatively, the container can wait to update the database with any updates to the bean instance as the container attempts to commit the transaction. The situation to consider in this case, however, is that such an update at commit time may result in a rollback of the transaction.

Entity Bean Concurrency with the Same Transaction Context

In the case in which an entity bean instance is accessed concurrently by clients and the same transaction context is associated with both client requests, this generally is considered an application error. However, in some situations it may be allowed. Such situations are sometimes referred to as “loopback” calls. Loopback calls are best described via a simple illustration as shown in Figure 28.2. Here we see an EJB client that invokes a method `foo()` on some entity bean A's interface. This call is then delegated to the bean A's instance, which in turn invokes a method `bar()` on some other entity bean B's interface. That call is delegated to entity bean B's instance, which in turn calls some method `naboo()` on entity A's interface and is then delegated to that bean's instance. This situation, when occurring within the same transaction context, is referred to as a loopback call because entity bean A's instance has in effect been invoked twice within the same transaction.

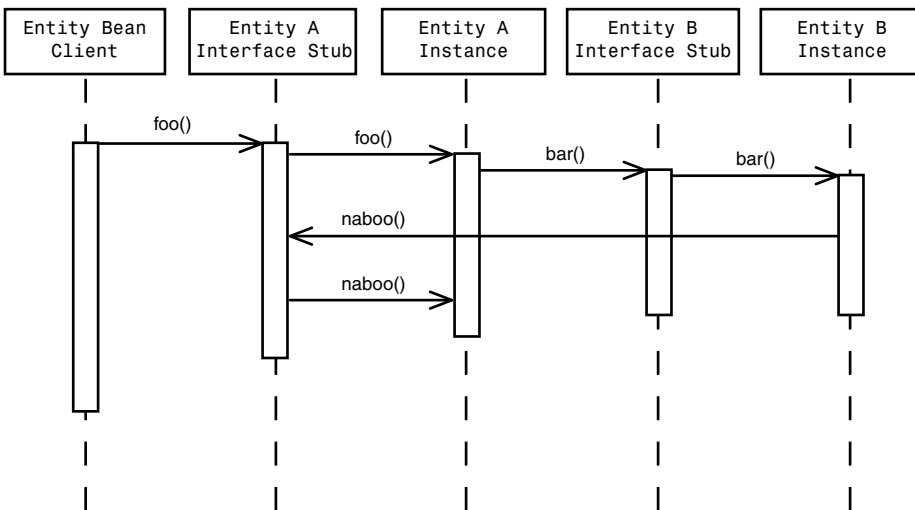


Figure 28.2 EJB loopback calls.

Such a scenario, of course, happens only by design of your entity beans. If you've designed your beans in such a way that this situation is possible, you must also consider designing your beans to allow for such behavior without affecting the integrity and correctness of processing. If you cannot create your beans to properly handle this situation, but the situation can indeed occur, then your entity bean is considered "nonreentrant." When an entity bean is nonreentrant and indicated as such to the container, the container will throw a `RemoteException` to remote clients and an `EJBException` to local clients attempting to access the same bean instance while it is still processing another method invocation invoked inside the same transaction. If you have been able to design your bean to handle such loopback calls, your bean is considered reentrant and you can indicate to the container that it may handle such situations as a reentrant bean.

If such concurrent requests to the same bean instance come from clients in different and external processes, the container may elect to throw a `RemoteException` to the client regardless of whether the bean is reentrant. Alternatively, an EJB container environment may attempt to handle such a distributed situation by creating one instance of the entity bean in one container process and routing all requests to that one instance, by creating the instance in multiple container processes but always calling `ejbLoad()` and `ejbStore()` at the beginning and end of each business method to ensure synchronicity of data, or by perhaps using a distributed lock manager on instances across the container environment.

Bean-Managed Transaction Demarcation

Only session beans and message-driven beans can take advantage of programmatic bean-managed transaction demarcation through the use of a `javax.transaction.UserTransaction` object, as well as through JDBC objects retrievable from the container environment. Entity beans must use container-managed demarcation, described in the next section. Session and message-driven beans must indicate their bean-managed as opposed to container-managed transaction demarcation preference via setting the `<transaction-type>` element inside of its `<session>` or `<message-driven>` EJB deployment descriptor to a value of `Bean` (as opposed to `Container`). Bean-managed transaction demarcation enables beans to explicitly indicate where a transaction must begin, commit, and potentially roll back via calls to `UserTransaction` from within their methods. EJBs can obtain handles to `UserTransaction` objects from the EJB container by calling `getUserTransaction()` on its `SessionContext` or `MessageDrivenContext` object (as inherited from `EJBContext`).

Note

This section contains a few simple code snippets that can be found beneath the `examples/src/ejava/ejb/transactions` directory for the code associated with this chapter located as described in Appendix A, "Software Configuration." Such code snippets are merely for purposes of illustrating various transaction management approaches and JTA API usage from within EJBs.

Alternatively, an EJB can also obtain a handle to a `UserTransaction` object from the EJB container via a JNDI lookup using the name `java:comp/UserTransaction` as shown here:

```
// Acquire handle to JNDI context from container environment
Context jndiCtx = new InitialContext();

// Look up UserTransaction from container environment
UserTransaction transaction
    = (UserTransaction) jndiCtx.lookup("java:comp/UserTransaction");
```

Tip

Despite what the EJB specification indicates, some vendor implementations may register the `UserTransaction` handle under a different JNDI name such as via the name `javax.transaction.UserTransaction`.

With a `UserTransaction` object in hand, calls to `begin()`, `commit()`, and `rollback()` can be used to programmatically demarcate transaction boundaries as you saw in Chapter 14. Between the boundaries of a `begin()` and `commit()`, all JDBC calls and calls to other resource managers that need to occur in an atomic, consistent, isolated, and durable fashion should be implemented. Given the flat transaction model required of EJBs, after a transaction is begun in such a fashion, another transaction should not be started until the current transaction is committed or rolled back. If an EJB client propagates a transaction to a bean-managed entity bean, the container will suspend the transaction and resume the transaction for the client only after the bean returns from its method call.

Furthermore, whereas stateful session beans can keep a transaction uncompleted across multiple calls on the same bean instance, stateless session beans and message-driven beans must commit or roll back the transaction before the bean method in which it was invoked returns from the call. Those EJBs implemented using the timer services (that is, `TimedObject` beans) must also begin and commit any transactions within the scope of their `ejbTimeout()` method calls. If a transaction is begun before a prior one has been completed, a `javax.transaction.NotSupportedException` must be thrown by the container. If a stateless session bean, message-driven bean, or `ejbTimeout()` method does not commit before its transactional method is returned, the container must discard the associated instance, log an error, roll back, and throw an exception (`RemoteException` or `EJBException`).

As described in Chapter 14, a transaction's isolation level defines how and when changes made to data within a transaction are visible to other applications accessing that data. Manipulating transaction isolation levels from within an EJB is specific to a particular resource manager. For JDBC, the transaction isolation level can be gotten and set with calls to a `java.sql.Connection` object via `getTransactionIsolation()` and `setTransactionIsolation()` calls, respectively. Recall from the earlier chapters

that JDBC Connection objects are obtained from calls to `getConnection()` on `javax.sql.DataSource` objects, which are in turn retrieved from JNDI lookups using `<resource-ref>` defined names.

For example, we might attempt to demarcate transactions within a session bean that periodically handles resolving data between a BeeShirts.com orders database and a separate database used to store orders sent to suppliers as shown here:

```
public class MySessionEJB implements SessionBean {
    private SessionContext ejbContext;
    ...
    public void setSessionContext(SessionContext aCtx) {
        ejbContext = aCtx;
    }

    public void resolveOrders(String supplierName) {

        // Get UserTransaction
        UserTransaction transaction = ejbContext.getUserTransaction();

        try {
            // Get context and data sources and connections
            InitialContext jndiCtx = new InitialContext();
            DataSource ordersDatabase =
                (DataSource) jndiCtx.lookup("java:comp/env/jdbc/Orders");
            DataSource suppliersDatabase =
                (DataSource) jndiCtx.lookup("java:comp/env/jdbc/Suppliers");
            Connection ordersConnection = ordersDatabase.getConnection();
            Connection suppliersConnection =
                suppliersDatabase.getConnection();

            // Possibly set isolation level for a connection
            suppliersConnection.setTransactionIsolation(
                Connection.TRANSACTION_READ_COMMITTED);

            // Begin transaction
            transaction.begin();

            // Do work...
            // 1) Create JDBC statements from connections.
            // 2) Do a bunch of JDBC update-related work on statements.

            // If error occurs while doing work...
            transaction.rollback();

            // Else if all is OK when done...end Transaction
            transaction.commit();
        }
    }
}
```

```

        // Close any connection and statements
    } catch (Exception e) {
        // Error occurred while doing work...
        try {
            transaction.rollback();
        } catch (SystemException se) {
            se.printStackTrace();
        }
        e.printStackTrace();
    }
}
...
}

```

As an example of a stateful session bean that begins a transaction when an order is placed via one method and subsequently notifies the supplier in another method before the transaction is completed, we have the following:

```

public class MySessionEJBBean implements SessionBean {
    private SessionContext ejbContext;
    private DataSource ordersDatabase = null;
    private DataSource suppliersDatabase = null;
    private Connection ordersConnection = null;
    private Connection suppliersConnection = null;

    public void setSessionContext(SessionContext aCtx) {
        ejbContext = aCtx;
    }

    public void placeOrder(String orderID, Object orderInfo) {
        // Get UserTransaction
        UserTransaction transaction = ejbContext.getUserTransaction();

        try {
            // Get context and data sources and connections
            InitialContext jndiCtx = new InitialContext();
            ordersDatabase =
                (DataSource) jndiCtx.lookup("java:comp/env/jdbc/Orders");
            ordersConnection = ordersDatabase.getConnection();

            // Begin transaction
            transaction.begin();

            // Do work...
            // 1) Create JDBC statements from connections.
            // 2) Update order database.
        } catch (Exception e) {
            // Error occurred while doing work...

```

```

        try {
            transaction.rollback();
        } catch (SystemException se) {
            se.printStackTrace();
        }
        e.printStackTrace();
    }
}

public void notifySupplier(String supplierName, String orderID) {
    // Get UserTransaction
    UserTransaction transaction = ejbContext.getUserTransaction();

    try {
        // Get context and data sources and connections
        InitialContext jndiCtx = new InitialContext();
        suppliersDatabase =
            (DataSource) jndiCtx.lookup("java:comp/env/jdbc/Suppliers");
        suppliersConnection = suppliersDatabase.getConnection();

        // Possibly set isolation level for a connection
        suppliersConnection.setTransactionIsolation(
            Connection.TRANSACTION_READ_COMMITTED);

        // Do work...
        // 1) Create JDBC statements from connections.
        // 2) Do a bunch of JDBC update-related work on
        //     order and suppliers database statements.

        // If error occurs while doing work...
        transaction.rollback();

        // Else if all is OK when done...end Transaction
        transaction.commit();

        // Close any connection and statements
    } catch (Exception e) {
        // Error occurred while doing work...
        try {
            transaction.rollback();
        } catch (SystemException se) {
            se.printStackTrace();
        }
        e.printStackTrace();
    }
}
...
}

```

Container-Managed Transaction Demarcation

Session, entity, and message-driven beans can all take advantage of declarative container-managed transaction management services. Such services are automatically provided by the container environment based on information configured in the EJB's deployment descriptors. Container-managed transaction demarcation in fact is a requirement for entity beans. To use container-managed transactions, a session or message-driven bean must specify the value of `Container` in its `<transaction-type>` element within the `<session>` or `<message-driven>` element in an EJB deployment descriptor, but no such option even exists for entity beans.

Container Transaction Configuration (`<container-transaction>`)

The means by which a container determines how to demarcate transactions for a bean is via establishment of zero or more `<container-transaction>` elements within an `<assembly-descriptor>` element. Recall that this `<assembly-descriptor>` element can be optionally defined within the root `<ejb-jar>` element for an EJB module as discussed in Chapter 24, "EJB Basics." The top-level structure of the `<container-transaction>` element is defined as explained here:

- `<container-transaction>`: Associates a transaction attribute with one or more EJB methods with sub-elements that follow.
- `<description>`: Describes this transaction management specification (zero or more).
- `<method>`: Identifies particular method(s) for which this transaction management specification applies (one or more).
- `<trans-attribute>`: Specifies the transaction attribute for the method(s) on a particular EJB that are associated with this attribute. The valid types are `Required`, `RequiresNew`, `Mandatory`, `Supports`, `NotSupported`, and `Never`.

Method Deployment Configuration (`<method>`)

The `<method>` element is further decomposed to identify those methods to which the specification applies as shown here:

- `<method>`: Identifies particular method(s) for which a containing specification applies with sub-elements that follow.
- `<description>`: Describes the method identified by this `<method>` element (zero or more).
- `<ejb-name>`: Identifies the EJB name to which this specification applies.
- `<method-intf>`: Indicates the type of interface on the EJB to which this specification applies: `Home`, `Remote`, `LocalHome`, `Local`, or `ServiceEndpoint` (optional).
- `<method-name>`: Identifies the EJB method(s) to which this specification applies. If an asterisk (*) is used, this particular specification applies to all methods on the

EJB. If another container-transaction is specified for this EJB with a specific name, the properties specific to that method override those defined by an `*`.

- `<method-params>`: Used to specify fully qualified parameter types associated with an EJB method to which this specification applies (optional). Providing this information uniquely identifies a particular EJB method and overrides any generic specification for a particular group of overloaded methods or for the whole EJB. Contains one or more `<method-param>` elements that define the fully qualified type for a method parameter.

Thus, if an `*` is used for a method name, the element specification associated with this setting becomes the default for all methods associated with that bean. If a method name is provided, such a method name's associated element specification overrides the `*` setting and the element specification associated with this setting becomes the default for all methods of that name. If a method name and its fully qualified parameter types are defined, the associated element specification overrides any other settings for that method. The `<method-intf>` element is used in those cases in which a particular method and operation signature is identically defined on two or more of the bean's interfaces.

Methods identified by such elements have transactions managed for them by the container according to the specified transaction attribute type. A session bean using such declarative transaction specifications must have all of its business-specific remote interface methods associated with a particular transaction attribute. An entity bean needs both its business-specific remote interface methods and its home interface methods associated with a transaction attribute. A message-driven bean using such declarative transactions must have its messaging listener interface method associated with a particular transaction attribute. If no transaction attributes are specified for the EJB, such attributes must be defined either by default or manually at deployment time.

Container-Managed Demarcation Example

As an example, suppose that we've defined the following transaction-unaware bean for which we want to allow a container to manage transaction demarcation:

```
package ejava.ejb.transactions;

import javax.ejb.*;
import java.rmi.*;

public class MyEntityEJBean implements EntityBean
{
    public void myMethod(){ /* Do something...*/ }
    public void myMethod(String foo){ /* Do something...*/ }
    public void myMethod(String foo, Object bar) { /* Do something...*/ }

    public void myOtherMethod(){ /* Do something...*/ }
    public void myOtherMethod(String foo){ /* Do something...*/ }
    ...
}
```

Now suppose that during assembly time, we decide the following about this bean's transaction semantics:

- All remote methods should have a `Required` transactional attribute by default. That is, the container will demarcate a transaction if the client does not.
- The remote `myMethod(String, Object)` method should have a `RequiresNew` transactional attribute associated with it. That is, the container will suspend any client transaction and create its own for the bean.
- Both remote `myOtherMethod()` methods should have a `Mandatory` transactional attribute associated with them. That is, the container will not demarcate transactions, but requires that the client does.

Given such transactional semantics, we might define our `ejb-jar.xml` file for this bean according to the basic structure shown here:

```
<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
  version="2.1"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/ejb-jar_2_1.xsd">
  ...
  <enterprise-beans>
    ...
    <entity>
      <description>Transaction-unaware Bean</description>
      <display-name>MyEntityEJBean</display-name>
      <ejb-name>MyEntityEJBean</ejb-name>
      <home>ejava.ejb.transactions.MyEntityHome</home>
      <remote>ejava.ejb.transactions.MyEntity</remote>
      <ejb-class>ejava.ejb.transactions.MyEntityEJBean</ejb-class>
      <persistence-type>Container</persistence-type>
      <prim-key-class>java.lang.String</prim-key-class>
      <reentrant>false</reentrant>
      ...
    </entity>
  </enterprise-beans>

  <assembly-descriptor>

    <container-transaction>
      <method>
        <description>
          Required transaction for all methods as default.
        </description>
```

```

        <ejb-name>MyEntityEJBBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>

<container-transaction>
    <method>
        <description>
            Require new transaction for myMethod(String, Object).
        </description>
        <ejb-name>MyEntityEJBBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>myMethod</method-name>
        <method-params>
            <method-param>java.lang.String</method-param>
            <method-param>java.lang.Object</method-param>
        </method-params>
    </method>
    <trans-attribute>RequiresNew</trans-attribute>
</container-transaction>

<container-transaction>
    <method>
        <description>
            Mandatory transaction for myOtherMethod(...).
        </description>
        <ejb-name>MyEntityEJBBean</ejb-name>
        <method-intf>Remote</method-intf>
        <method-name>myOtherMethod</method-name>
    </method>
    <trans-attribute>Mandatory</trans-attribute>
</container-transaction>

</assembly-descriptor>

</ejb-jar>

```

Interface Calls with Container-Managed Demarcation Beans

Although bean implementations should not attempt to perform any transaction demarcation or isolation setting themselves under container-managed demarcation, beans can call `getRollbackOnly()` and `setRollbackOnly()` on their `EJBContext` object. `EJBContext.setRollbackOnly()` is used to set the current transaction state such that it can only roll back and never commit. The `EJBContext.getRollbackOnly()`

method can be called to return a `boolean` value indicating whether such a state was already set. Setting a transaction to roll back is useful only to inform the container that an application error has occurred within the current transaction, and that the state of the system affected by this transaction should thus be considered invalid. The container will then roll back the transaction when it is appropriate.

Stateful session beans can also be made a tad more transaction aware when they implement the `javax.ejb.SessionSynchronization` interface as shown in Figure 25.2 of Chapter 25, “Session EJB.” Stateful session beans that implement the `SessionSynchronization` interface’s `afterBegin()`, `beforeCompletion()`, and `afterCompletion(boolean)` methods can be informed when the container has just begun a transaction, is about to complete a transaction, or has just completed the transaction with a `true` status if it committed, respectively.

Vendor-Specific Transaction Demarcation

On a final note, vendors will also define their own specific transaction semantics for EJBs. For example, the BEA WebLogic Server’s `weblogic-ejb-jar.xml` deployment descriptor contains a `<transaction-descriptor>` element that contains a `<trans-timeout-seconds>` element to define the maximum time in seconds a transaction may exist between its beginning and completion. If this time expires before the transaction can be completed, the transaction will be rolled back.

Vendor-specific deployment descriptors may also define a transaction isolation level in a vendor-specific fashion. BEA WebLogic’s `weblogic-ejb-jar.xml` deployment descriptor, in fact, defines such information inside of a `<transaction-isolation>` element according to the following basic structure with isolation levels defined as in Chapter 14:

```
<transaction-isolation>
  <isolation-level>TransactionSerializable</isolation-level>
  <method>
    <description>
      Describe the method to which this isolation
      level applies here.
    </description>
    <ejb-name>MyEntityEJBean</ejb-name>
    <method-intf>Remote</method-intf>
    <method-name>myMethod</method-name>
    <method-params>
      <method-param>java.lang.String</method-param>
      <method-param>java.lang.Object</method-param>
    </method-params>
  </method>
</transaction-isolation>
```

J2EE and EJB Availability and Scalability

EJB-based applications may represent large-scale enterprise applications encapsulating business logic and operations that are crucial to an enterprise's health, success, and growth. Chapter 17 discussed some of the reasons enterprise applications often need to be highly available and scale to meet the demands of users. EJB-based applications are where some of the more critical needs for scalability and availability are required. This section discusses just a few facets of this complex problem and how to apply solutions to EJB-based applications.

Redundant Server Overview

The core principle involved with providing highly scalable and highly available enterprise applications is the principle of redundancy. Redundant servers involve running more than one physical server instance in an operational environment. Redundant servers may be used for two primary purposes affecting the scalability and availability of your system:

- *Load Balancing:* Requests from clients may be distributed in some fashion across multiple server instances. This helps ensure that excessive client load doesn't reduce the throughput of your servers and thus affect availability by virtue of reduced capability to satisfy client demands.
- *Fail-over:* In the event of a failure of one server process, a means to fail-over to another server process is provided with minimal to no disruption to the client's session.

In J2EE environments, multiple types of objects may need to be made redundant in order to achieve the goals of load balancing and fail-over. These include J2EE components such as EJB, Java Servlet, and JSP components. These may also include J2EE resources such as JMS, JDBC, and distributed communication resource connections. The remainder of this section discusses load balancing and fail-over management generally in J2EE environments and then focuses on its more complicated application to EJB servers.

J2EE Clusters

Figure 28.3 depicts a basic redundant unit associated with J2EE servers. This unit is only one example of a J2EE cluster. A J2EE cluster, as we define it here, is basically a collection of physical servers that are aware of one another in some way such that they can expose themselves as a high-assurance platform used by distributed clients. Clients communicate with the servers as a unit and generally remain unaware of the fact that they are talking to a clustered server environment. From the client's perspective, they are communicating with a server process that just happens to rarely if ever "go down." As shown in the diagram, every candidate physical tier of the clustered environment is replicated and redundant. Any number of servers within each tier, of course, may be built out so that a particular tier could have two, three, or more servers. Furthermore, the diagram

of Figure 28.3 simply intends to convey the core components of a highly available and scalable J2EE cluster. The services offered by each tier (to be discussed next) in fact may be collapsed and collocated within the same physical process.

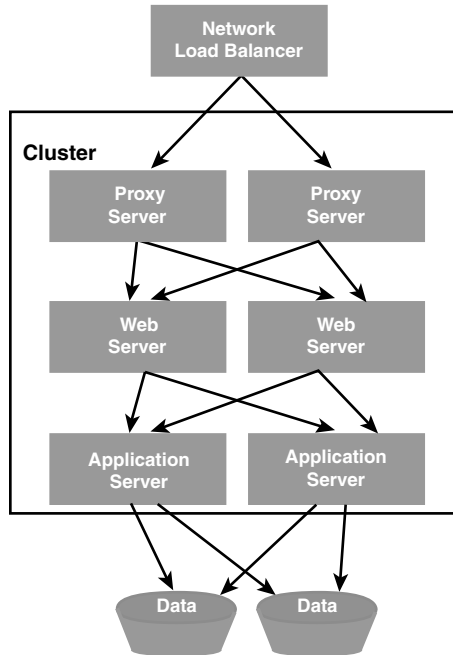


Figure 28.3 J2EE cluster.

J2EE EJB and Web Servers in a Cluster

Although it is not a requirement, physically separate J2EE Web server and EJB tiers may be employed by such a clustered architecture environment. In such a configuration, Java Servlets and JSP components run inside of Web server processes to expose themselves to Web clients, whereas EJBs run inside of application server processes to expose themselves to the Web tier and perhaps to thick clients operating inside the enterprise firewall. Such an approach provides demilitarized zones (DMZ) that keep access to the database secured behind the wall of the application server tier.

Proxy Servers in a Cluster

Additionally, a proxy tier may be provided inside of a J2EE cluster. These servers act as lightweight Web Servers whose primary function is to provide another DMZ and perhaps make use of a special plug-in to load balance across J2EE Web servers, as well as pin requests to particular servers. Such a plug-in may be needed to inform the proxy server

how to read and recognize session and server information baked into URLs or cookies. The server identifiers baked into such URLs or cookies indicate which back-end Web server a particular request is to be routed to. If a request comes in and the proxy server reads the URL or cookie and understands that it should pin the request to a particular server, and that server happens to be down, then the proxy server will route the request to one of the backup servers perhaps identified in the request's URL or cookie.

Of course, a particular J2EE vendor's offering may also offer other configurations whereby proxy, Web, and application servers may be co-located as a collapsed redundant tier. Or the servers might be configured in such a way that they share all of their state with all other servers in their tier of the cluster. In such a way, a proxy server would not require any special plug-in to determine how to route a particular request in the event of a failure. Rather, the proxy server would be able to route the request to any Web server in the cluster because it can be assured that all servers in that cluster have shared their state and can act as a "hot" backup server for any client. The disadvantage of such a configuration, however, is that there would need to be increased communications overhead between the servers in a tier to ensure that all servers in that tier have replicated state of one another. In the situation in which the proxy servers use a special plug-in to read URLs and cookies, they are able to know which specific servers act as backups for other servers, and hence the need for replicated state across an entire cluster tier is not needed. Another approach would be to have the servers define one particular cluster policy indicating which servers act as backups for which servers and where to route requests in the event of failures as a general policy.

Network Load Balancers in a Cluster

Also depicted in Figure 28.3 is the concept of a network load-balancing mechanism. In high-assurance operational Web environments, a front-end means to balance incoming requests across multiple proxy servers may be employed. This is because proxy servers, though relatively simple servers, are still prone to failure due to the inherent complexity of processing software on a general computing platform. Hence, a hardware-based product that balances incoming requests across proxy servers may be employed. Many network load balancers operate by examining low-level source and destination IP address headers to determine how to allocate requests across servers.

In high-assurance operational Web environments, a single IP address is used to indicate the target address for a clustered server environment. The network load balancer is configured to map this address to the target proxy server addresses. The network load balancer then picks that server, rewrites the IP destination address header, and delivers the request to the selected proxy server. Any subsequent requests may be pinned to that selected proxy server based on the incoming IP address. If the proxy server fails, the IP load balancer may route the request to another proxy server. Alternatively, the IP load balancer may simply load balance requests indiscriminately across the bank of proxy servers without any session pinning.

One issue may arise if the incoming address is changed in mid-session for the client. This may happen if the client requests are coming through a particular ISP that happens

to balance client sessions itself across different servers. Thus, for example, if a client request is coming through a particular ISP and its IP address (from the IP load balancer's perspective) is one value at one particular instant, the ISP may change that address and it then looks like another potentially new client session. Hence the network load balancer may send that request to a different proxy server. In the case of Figure 28.3, this is not a problem because the proxy servers simply use the information baked into URLs or cookies to determine how to route requests to the intended servers. When multiple clusters are used, however, a request may be routed by the IP load balancer to a server in a different cluster. Because it is presumed that redundancy communication between servers occurs only within the confines of a cluster, this would result in a lost client session.

Multiple J2EE Clusters

Multiple J2EE clusters may be needed in situations in which a site needs to operate at nearly 100% availability. In these scenarios, to update software on the server, to upgrade versions of Web and application servers, or perhaps to upgrade a database server in some way, particular servers may need to be brought down to perform these updates and upgrades. Multiple clusters can be used such that the sessions in one cluster are slowly “bled” from the servers by telling the network load balancer to stop allocating new requests to the proxy servers in one of the clusters. After all the sessions in that cluster are finished, the servers in that cluster may be shut down and upgraded. While this is occurring, the servers in the other cluster may be servicing requests.

As was indicated, network load balancers may route requests to different proxy servers if the incoming IP addresses change. Of course, if a different target address and port combination is received (such as the case when switching between secure and nonsecure ports), then the network load balancer may also route the request to a different proxy server. For one, session pinning must be supported by the network load balancer to even allow for this functionality to pin requests to particular proxy servers by the network load balancer. However, in the anomalous situations of changing source and destination addresses, the network load balancer can still route the requests to different servers in different clusters. Particular design approaches may be undertaken to resolve this situation. One might create a Web server plug-in or J2EE Web component handler that redirects the first request from a Web client back to a host name associated with the cluster as a whole. The network load balancer then routes requests to that host name to those proxy servers associated with that cluster.

J2EE Web Component Clustering

Clustering Web servers can be rather straightforward. After configuring any network load balancing and proxy tier servers, the J2EE Web servers run as they would in a standalone fashion, except for a few additional configuration parameters. The state that must be preserved in order to enable Web tier clustering is luckily largely localized to one general physical area: the HTTP session. Because of such localization and because most Web designs (at least those designed well) don't store a lot of information inside of HTTP

session memory on the server, it is relatively easy to make such information redundant. This is often accomplished in one of two basic ways:

- *HTTP Session Backup*: Whenever state is written to an HTTP session object, at least one backup server for the Web server is communicated this state update. In the event of a failure, the backup server is used to host the client's session.
- *Persisted HTTP Sessions*: Whenever state is written to an HTTP session object, this state is persisted in some way. In the event of a failure, the persisted form of the session is read by a backup server.

The efficacy and performance of how such clustering occurs will be a function of your particular vendor's server and configuration options. For example, if your vendor requires that all servers in a tier be broadcast the session updates, this may be inefficient if your cluster architecture employs a large number of servers and hence requires an excessive amount of inter-server communications. If you use an approach whereby session information is persisted, you'll want to consider how such information is persisted and how often the information is persisted in order to assess whether an extraordinary amount of overhead will be involved with such operations. Regardless, the basic Web tier clustering options available to you, the different policy options and parameters for such clustering available to you, and your particular Web tier design all must be taken into consideration on a vendor-specific and design-specific basis.

J2EE EJB Clustering

Configuring servers to cluster on a Web tier is relatively straightforward compared to the options available to you in the application server tier. Because EJBs are often longer-lived and coarser-grained components than Web tier components, their operational demand is greater. This section discusses the basic options often available to a developer in vendor-provided EJB clustering environments. Not all options discussed in this section are available in all vendor EJB server configurations. This is because such features are not required by the J2EE or EJB standards. However, this section does aim to arm you with the most common features available and concepts involved so that you can more rapidly hone in on how to employ such features with your particular EJB application server.

General Design Considerations

In the event of a failure, the state that is contained by EJBs may be more likely to be preserved if the client also held that state or if it was persisted in some way. That is, when using value objects to pass information between EJB client and server, the client may have references to such value objects and utilize them to re-issue a request to another server in the event of a failure. If the state inside an EJB is persisted and a failure arises, the client may also re-issue a request to an EJB that can reconstruct its state prior to the failure.

Although such design guidelines are common sense, it doesn't really much help the developer who wants to assume that he's using a server environment that is highly

assured in a fashion transparent to the client. Furthermore, the EJBs themselves must also be designed with such value object or state persistence logic in mind.

Cluster-Aware EJB Interface Stubs

When clustering EJBs, generally some form of special stubs needs to be used by EJB clients so that they know how to communicate with the clustered server environment. The stubs may be distinguished as to whether they are stubs for home interfaces or stubs for remote object interfaces. Because clustering functionality is implemented in a vendor-specific fashion, a vendor-specific tool must be provided to generate such cluster-aware stubs. Hence, EJB clients that communicate with clustered EJBs need to be provided with such stubs. Because EJB utilizes RMI underneath the hood, as discussed in Chapter 9, “RMI Communications,” such stubs may be dynamically downloaded by the client. The EJB client does not need to be recompiled to work with such stubs because the standard EJB client interface may be used with a cluster-aware stub being referenced transparently to the EJB client.

Home interface stubs that are cluster-aware must contain information that is used to determine to which server in a cluster a particular home interface operation is to be directed. That is, when an EJB create, remove, finder, or home business method is invoked, a means for directing that request to an appropriate server in the cluster must be baked into the logic of the home interface stub.

For load-balancing purposes, the stub may be constructed to route a home interface request to a different server in the cluster according to a particular load-balancing policy baked into the stub. Vendors offering different policies for load balancing generally allow the specification of which load-balancing policy is to be employed on a per-EJB-type basis in a vendor-specific deployment descriptor. Of course, whether or not a particular home interface type is clusterable should also be configurable via vendor-specific deployment descriptors. If a vendor allows for an extensible way to define new load-balancing policies, a GUI interface may be provided or custom class extension mechanism may be provided that allows for you to specify your own parameters and algorithms used in the selection of a server used when load balancing. The more common load-balancing policies that may be provided to you or that you may build include the following:

- *Round robin*: Servers are selected one after the other in a cyclic list (for example, server-1, server-2, server-3, server-1, server-2,...).
- *Random*: Servers are selected on a random basis.
- *Weighting*: Servers are weighted according to how often they should receive requests according to a random selection distribution (for example, server-1 50% of the time, server-2 10% of the time, server-3 40% of the time).
- *Affinity*: Server selection preferences are determined according to whether the client already has a connection established with that server.

Home interface stubs that are aware of multiple servers in a cluster for load-balancing purposes may also use such information for fail-over purposes. Thus, if a particular server is attempted to be accessed via a home interface invocation and that server does not respond, the home interface stub may indicate that such a server has failed and direct the request to another server in the cluster. The stub may also be implemented to transparently request updated information from a server in a cluster regarding which servers have failed and which have come alive. This information may then be used to update the list of available servers to access inside the client's interface stub.

After a home interface is used to create an instance of an object on one of the servers in a cluster, a special remote object stub implementation associated with that EJB may also be returned to the EJB client. This cluster-aware remote interface stub can also be used for load-balancing and fail-over purposes. The particular nature of such is a function of a particular EJB type.

Clustered Session Beans

The interface stub for a stateless session bean can route requests to any server in the cluster. Different requests to the stateless session interface by the client can be routed to different servers in the cluster. This is because the state of a stateless session bean is not bound to a particular EJB client session. Of course, in addition to the load-balancing opportunity such an interface stub type allows, another server may be selected for a stateless session bean interface request if the original server to which a request was directed failed.

Stateful session beans must have their state preserved for a particular EJB client's session with the bean. The means by which such support is provided may be similar to that provided for HTTP session preservation with Web components. That is, a stateful session bean may be clustered this way:

- *Stateful Session Backup*: Whenever state is updated in a stateful session bean instance, one or more backup servers for the Web server are communicated this state update. In the event of a failure, the backup server is used to reconstruct a stateful server bean instance with this state.
- *Persisted Stateful Sessions*: Whenever state is updated in a stateful session bean instance, this state is persisted in some way. In the event of a failure, the persisted form of the stateful session bean's state is read by a backup server and used to reconstruct a new stateful session bean instance.

The decision for how and when to replicate the state to a backup session bean or storage is up to the EJB container vendor. Generally, after a transaction, state should be updated to the backup session bean state. For nontransactional calls, the state should be updated after a method invocation. In the rare event of a failure after primary state is updated but before backup state can be updated, or in the event of a failure of both primary and backup state preservation mechanisms, the integrity of a stateful session bean may be compromised.

Clustered Entity Beans

Entity beans have advantages in that their state is persisted to a database, transactional, and hence more easily reconstructed. EJB container vendors may allow you to configure the nature of entity bean clustering and persistence frequency with the database, however. The more common entity bean clustering policies include the following:

- *Read-Write Entity Beans:* An interface stub for the bean is pinned to a particular server containing the associated entity bean instance. The bean's `ejbLoad()` method is invoked to load the bean's state from the database before each transaction is begun and invoke the `ejbStore()` method to store the bean's state to the database on commit of a transaction.
- *Read-Only Entity Beans:* An interface stub for the bean load balances across entity bean instances in multiple servers across the cluster. Hence, the state associated with such beans is cached inside every server in the cluster.

EJB Idempotence

Although all such clustering approaches work well for fail-over situations whereby a failure occurs between method invocations, if a failure occurs during a method invocation, then special considerations must be made. If a bean is designed to be “idempotent,” this essentially means that multiple method invocations can be made on the bean any number of times and not result in any updates made to persistent or cached state of the bean that affects the behavior of the application. You simply have to think about the case when a bean has one of its methods invoked more than once. If the two invocations affect the state of the system in such a way that would be different if the method were invoked only once, then the bean's method is not idempotent.

For example, in the case of a shopping cart, when we add items to the cart, clearly the methods to insert an item into the cart change the state of the system and hence multiple invocations are different than a single invocation. Thus, such a method is not idempotent. If a bean is a simple content browser that reads product items from the database, multiple invocations of the bean do not result in state updates, thus allowing us to classify such a bean's associated methods as idempotent.

Some vendors require that you indicate whether a particular bean is idempotent because that information will allow the container to manage how to fail-over in those situations in which a failure occurs during processing of a particular method. A vendor may also allow the specification of idempotence at the level of an entire bean and/or individual method level. In some cases during such failure types involving non-idempotent beans, nothing can be done and the EJB client simply needs to be made aware of the fact, perhaps via some exception to handle.

EJB Security

As with any distributed object used in security-critical enterprise applications, EJBs must be secured. However, EJB components operate inside of a container environment and

rely on the container to provide distributed connectivity to an EJB, to create and destroy EJB instances, to passivate and activate EJB instances, to invoke business methods on EJBs, and to generally manage the life cycle of an EJB. Because such control is relinquished to an EJB container/server environment, securing the EJB also relies heavily on the support provided by the EJB container/server environment. Security mechanisms can be distinguished by standard mechanisms required by the J2EE and EJB specifications, mechanisms that are EJB container/server vendor-specific, and mechanisms that may be hand-coded by the EJB developer.

Figure 28.4 illustrates the basic architecture required for securing EJBs. Standard security mechanisms defined for EJBs are currently largely focused around providing a minimal set of constructs for role-based EJB access control. Standard mechanisms for determining role-based permissions to access EJB methods may be tapped by EJB components programmatically via a few interfaces to the EJB container context as exposed by the EJB API. Standard EJB method access control mechanisms can also be defined declaratively via a set of standard XML elements contained in a standard EJB deployment descriptor. Additionally, a few vendor-specific access control features are needed to support the mapping of security roles defined in standard deployment descriptors to principal identities managed by the operational environment.

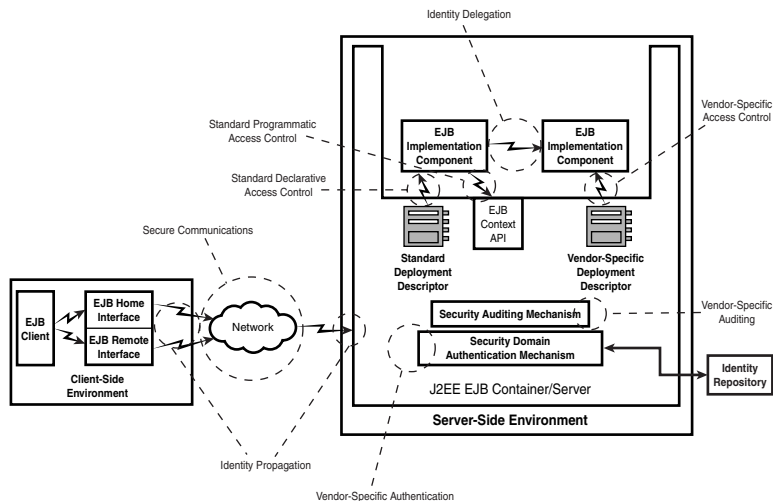


Figure 28.4 EJB security architecture.

Although a mechanism for authenticating a user is implied by virtue of the fact that a particular user must be granted access to a particular EJB method, no standard means for authenticating a user is explicitly required in the EJB v2.1 and the J2EE v1.4 specifications. However, the JAAS, now part of the underlying J2SE v1.4, represents the primary option for implementation of authentication in a standard fashion. Nevertheless, some

form of identity propagation from client-to-server and principal authentication with an identity repository must be provided by an EJB container/server vendor for practical EJB enterprise security. The policy for delegating principal identity between EJBs must also be defined by an EJB container/server vendor. Furthermore, the security of the communications session between EJB client and server must be provided by a vendor in an interoperable fashion for practical enterprise deployment scenarios. Vendors may also provide a means for auditing security-critical EJB events.

Standard Programmatic EJB Access Controls

Although the programmatic implementation of security access control logic within EJB components is discouraged in favor of container-managed security features by the EJB specification, in many practical cases it may be inevitable. A minimal set of standard methods is provided by the EJB API to enable programmatic access control from within your EJB. As illustrated in Figure 28.5, two primary EJB hooks for obtaining security information from the EJB container environment are provided by the `javax.ejb.EJBContext` object (with two other EJB v1.0 methods deprecated). Other non-security-related `EJBContext` methods are not shown in this diagram. Note that message-driven beans are not shown in this diagram because message-driven beans do not have a client security context. The container in fact throws an `IllegalStateException` if a message-driven bean attempts to invoke the methods defined on the `EJBContext` shown in Figure 28.5.

A handle to an `EJBContext` object will be available to an EJB implementation object when the EJB container sets the context object onto a bean instance after the bean instance has been created by the container. For session bean implementations, such as `MySessionBean` shown in Figure 28.5, the EJB container will call `setSessionContext()` on the bean with a `javax.ejb.SessionContext` object. For entity bean implementations, such as `MyEntityBean` shown in Figure 28.5, the EJB container will call `setEntityContext()` on the bean with a `javax.ejb.EntityContext` object. After a context is set onto a bean, security-related calls to the `EJBContext` object are callable only from within the context of a business-specific method on the EJB. Otherwise, a `java.lang.IllegalStateException` will be thrown by the container to indicate that no security context exists.

The `EJBContext.getCallerPrincipal()` method is invoked by an EJB to obtain a handle to a `java.security.Principal` object. The `Principal` represents the particular principal identity on behalf of which the invoking EJB client is acting. A call to `Principal.getName()` by the bean can return a `String` object that can be used for business-specific security checking logic decision making. The deprecated `EJBContext.getCallerIdentity()` method may still be supported by your container, but a vendor may choose to throw an exception or return `null` from this method.

The `EJBContext.isCallerInRole(String)` method is used to ask the EJB environment whether the current principal associated with this security context is a

member of the role passed in as a `String` to this method. A `boolean` return value indicates whether the caller is indeed in this role. The deprecated `EJBContext.isCallerInRole(Identity)` method may still be supported by your container, but a vendor may choose to throw an exception from this method.

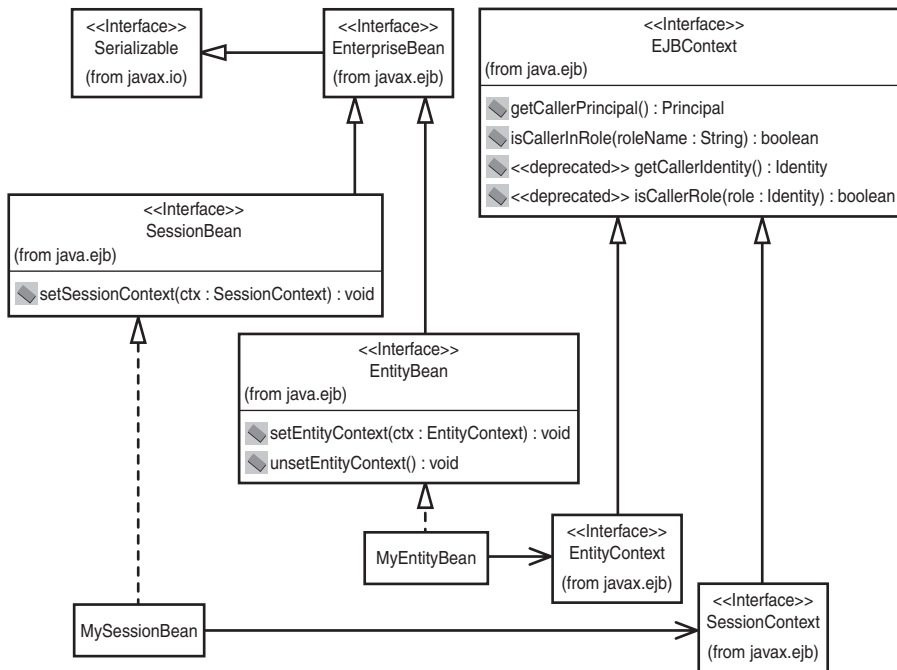


Figure 28.5 EJB security APIs.

Note

This section contains a few simple code snippets as well as working example code that can be found beneath the `examples/src/ejava/ejb/security/basic` directory for the code associated with this chapter and located as described in Appendix A. Note that the examples in that directory related to bean-managed programmatic security are simply snippets used for illustrative purposes in the text that follows. The examples related to container-managed declarative security for this chapter are working code examples integrated with the EJB code of Chapter 25 (and that chapter's dependent code). Hence, you will need the code from Chapters 19, 25, and 26 to run this chapter's declarative security examples.

As an example, the `SecureOrderManagerEJBean` class of Listing 28.1 extends the `ejava.ejb.session.ordermgr.OrderManagerEJBean` discussed throughout our EJB examples in previous chapters. Here we see that the

SecureOrderManagerEJBBean overrides the `queryOrders()` method to perform programmatic security checking before it allows the client to invoke the method.

Listing 28.1 **Programmatic EJB Access Control**
(SecureOrderManagerEJBBean.java)

```

package ejava.ejb.security.basic;
...
public class SecureOrderManagerEJBBean extends OrderManagerEJBBean {

    /**
     * Get customer username (i.e., email) based on customer ID
     */
    private String getUserName(int customerID){
        String userName = "";
        try{
            CustomerLocal customer
                = getCustomerHome().findByPrimaryKey(new Integer(customerID));
            userName = customer.getEmail();
        }catch(Exception e){
            System.out.println("Username not found " + e.getMessage());
        }
        return userName;
    }

    /**
     * Query for the orders in the DBMS and return them.
     */
    public Vector queryOrders(int customerID) {
        Vector orders = null;
        try {
            // Get a username associated with this customer ID
            String customerUserName = this.getUserName(customerID);

            // Get the EJB context for this bean
            EJBContext ejbContext = this.getSessionContext();

            // Get principal from context and principal name
            java.security.Principal caller = ejbContext.getCallerPrincipal();
            String callerUserName = caller.getName();

            // If not in manager role and not the user associated with
            // this order, then deny access...
            if ((ejbContext.isCallerInRole("manager") == true)
                || (callerUserName.equalsIgnoreCase(customerUserName))) {

```

Listing 28.1 Continued

```

        // Allow access to do security-critical stuff here...
        // ...which in this case is to call superclass method
        orders = super.queryOrders(customerID);
    } else {
        // Access denied. Audit this event and return...
        System.out.println("**** ILLEGAL ACCESS ATTEMPT ****"
            + "\n OrderManager.queryOrders(int)"
            + "\n Caller Name; " + callerUserName
            + "\n Customer Name " + customerUserName);
    }
} catch (Exception e) {
    e.printStackTrace();
}
return orders;
}
}

```

The `queryOrders()` method first looks up a username for a given customer ID which essentially returns the email address for that customer used as the customer's username. Calls to the EJBContext security methods are then performed to retrieve the username for the user attempting to call this bean's method. If the caller username matches the username for the customer whose order is being examined, or if the caller is in the manager role, then the superclass's unprotected `queryOrders()` method is invoked. Otherwise, the event is audited as a security-critical event.

EJB Security Role References (<security-role-ref>)

Whenever a call to `EJBContext.isCallerInRole()` is made from within EJB code, an associated `<security-role-ref>` should be identified in the EJB's standard deployment descriptor for that bean. The `<security-role-ref>` element is defined within an `<entity>` element for entity beans and within a `<session>` element for session beans. The `<security-role-ref>` element has the following definition:

- `<security-role-ref>`: Identifies a security role reference for the containing EJB with sub-elements that follow.
- `<description>`: Describes this security role reference (zero or more).
- `<role-name>`: Identifies a logical role name that this EJB uses.
- `<role-link>`: References a role defined in a `<security-role>` element.

As an example, if we have defined a standard `ejb-jar.xml` file for our `SecureOrderManagerEJB` session bean that implements the `queryOrders()` method as defined in Listing 28.1, then we would want to define a `<security-role-ref>` entry for the referenced `Managers` role as illustrated in Listing 28.2.

Listing 28.2 EJB Security Role Reference

```

<ejb-jar...>
  ...
  <enterprise-beans>
    <session>
      <display-name>SecureOrderManagerEJB</display-name>
      <ejb-name>SecureOrderManagerEJB</ejb-name>
      <home>ejava.ejb.session.ordermgr.OrderManagerHome</home>
      <remote>ejava.ejb.session.ordermgr.OrderManager</remote>
      <ejb-class>
        ejava.ejb.security.basic.SecureOrderManagerEJBBean
      </ejb-class>
      <session-type>Stateless</session-type>
      ...
      <security-role-ref>
        <description>This bean references Managers role.</description>
        <role-name>manager</role-name>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>

```

It is the responsibility of EJB developers to define in a deployment descriptor those security roles that their EJB implementations programmatically reference. However, it is up to the EJB assembler and deployer to map such roles to security roles and users in the operational deployment environment. The next section illustrates how such a mapping occurs in the context of standard declarative EJB access controls.

Standard Declarative EJB Access Controls

Standard declarative EJB access control mechanisms are defined as XML elements in the standard EJB deployment descriptor file. In addition to the `<role-name>` element, a `<role-link>` element may also be defined within an EJB's `<security-role-ref>` element. This element value is defined during EJB assembly to reference a role name specified by an individual (that is, EJB assembler) cognizant of the security roles assumed by a particular deployment environment. Thus, an EJB assembler might modify the standard `ejb-jar.xml` file to map a programmatic role name identified by the `<role-name>` element to an assembly-specific role name identified by a `<role-link>` element.

As an example, our `SecureOrderManagerEJBBean` bean's deployment descriptor may be modified to incorporate an assembly-specific `<role-link>` element as illustrated in Listing 28.3.

Note

Because Listing 28.2 and Listing 28.3 are simply illustrative examples, the actual example deployment descriptor is named `ejb-jar-bean.xml` in the `examples/src/ejava/ejb/security/basic` directory.

Listing 28.3 **EJB Security Role Reference with Role Link** (`ejb-jar-bean.xml`)

```
<ejb-jar...>
  ...
  <enterprise-beans>
    <session>
      <display-name>SecureOrderManagerEJB</display-name>
      <ejb-name>SecureOrderManagerEJB</ejb-name>
      ...
      <security-role-ref>
        <description>This bean references Managers role.</description>
        <role-name>manager</role-name>
        <role-link>Managers</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

EJB Security Role Declarations (`<security-role>`)

The `<role-link>` element within an EJB's `<security-role-ref>` element may be defined during EJB assembly to refer to a `<role-name>` defined within a particular `<security-role>` element. All logical security roles defined for a particular EJB module are identified by `<security-role>` elements that sit within an `<assembly-descriptor>` element, which in turn can optionally be defined within the root `<ejb-jar>` element for an EJB module. The `<security-role>` element is defined as shown here:

- `<security-role>`: Identifies those security roles defined for an EJB module with sub-elements that follow.
- `<description>`: Describes this logical security role (zero or more).
- `<role-name>`: Identifies a logical role name that this EJB module uses.

As an example, an EJB assembler would define a `<security-role>` element for the Managers role linked by our `SecureOrderManagerEJB` bean as illustrated in Listing 28.4.

Listing 28.4 **EJB Security Role** (ejb-jar-bean.xml)

```

<ejb-jar...>
  ...
  <enterprise-beans>
    <session>
      <display-name>SecureOrderManagerEJB</display-name>
      <ejb-name>SecureOrderManagerEJB</ejb-name>
      ...
      <security-role-ref>
        <description>This bean references Managers role.</description>
        <role-name>manager</role-name>
        <role-link>Managers</role-link>
      </security-role-ref>
    </session>
  </enterprise-beans>

  <assembly-descriptor>
    <security-role>
      <description>
        Only managers can have this role.
      </description>
      <role-name>Managers</role-name>
    </security-role>
  </assembly-descriptor>
</ejb-jar>

```

Although such `<security-role>` elements can be defined to link from `<security-role-ref>` elements defined for EJBs that use programmatic EJB security features, these `<security-role>` elements can also be defined for EJBs that make exclusive use of declarative EJB security features. That is, EJBs using declarative EJB security features do not rely on programming security checks as the `SecureOrderEJBBean` of Listing 28.1 demonstrated. Rather, an EJB can solely rely on specifying security properties in the EJB's deployment descriptors. This is exemplified in Listing 28.5 by redefining the `ejb-jar.xml` file for our original `OrderManagerEJBBean` class having no programmatic security features.

Listing 28.5 **EJB Declarative Security** (ejb-jar.xml)

```

<?xml version="1.0" encoding="UTF-8"?>
<ejb-jar
  version="2.1"
  xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
  http://java.sun.com/xml/ns/j2ee/ebj-jar_2_1.xsd">

```

Listing 28.5 Continued

```

<display-name>OrderManagerEJB</display-name>
<enterprise-beans>
  ...
  <session>
    <display-name>OrderManagerEJB</display-name>
    <ejb-name>OrderManagerEJB</ejb-name>
    <home>ejava.ejb.session.ordermgr.OrderManagerHome</home>
    <remote>ejava.ejb.session.ordermgr.OrderManager</remote>
    <ejb-class>
      ejava.ejb.session.ordermgr.OrderManagerEJBean
    </ejb-class>
    <session-type>Stateless</session-type>
    ...
    <security-identity>
      <description>
        Use caller identity in access control decisions
      </description>
      <use-caller-identity/>
    </security-identity>
  </session>
</enterprise-beans>
<relationships>
  ...
</relationships>
<assembly-descriptor>

  <security-role>
    <description>
      Every authenticated user can have this base role.
    </description>
    <role-name>EveryUser</role-name>
  </security-role>

  <security-role>
    <description>
      Only managers can have this role.
    </description>
    <role-name>Managers</role-name>
  </security-role>

  <method-permission>
    <description>
      Managers can invoke all Order Manager methods by default.
    </description>
    <role-name>Managers</role-name>

```

Listing 28.5 **Continued**

```

    <method>
      <ejb-name>OrderManagerEJB</ejb-name>
      <method-name>*</method-name>
    </method>
  </method-permission>

  <method-permission>
    <description>
      No access control is required to
      create a handle to an OrderManager.
    </description>
    <unchecked/>
    <method>
      <ejb-name>OrderManagerEJB</ejb-name>
      <method-name>create</method-name>
    </method>
  </method-permission>

  <method-permission>
    <description>
      Every authenticated user can create a new order.
    </description>
    <role-name>EveryUser</role-name>
    <method>
      <ejb-name>OrderManagerEJB</ejb-name>
      <method-name>createNewOrder</method-name>
    </method>
  </method-permission>
  ...
  <exclude-list>
    <description>
      These methods should not be accessed at all.
    </description>
    <method>
      <ejb-name>OrderManagerEJB</ejb-name>
      <method-name>getCustomer</method-name>
    </method>
  </exclude-list>
</assembly-descriptor>
  ...
</ejb-jar>

```

EJB Method Permissions (<method-permission>)

Listing 28.5 demonstrates two security roles being defined for our application. The `Managers` and `EveryUser` roles respectively correspond to those users who are managers and are authenticated users of a system. Special deployment descriptor elements can then be defined that dictate those security roles that can access particular methods on an EJB. Zero or more `<method-permission>` elements defined within an `<assembly-descriptor>` element are used to provide such role-to-method access control mappings. A `<method-permission>` element is defined as shown here:

- `<method-permission>`: Maps methods on an EJB to the roles allowed access to such methods or indicates that such methods are to be unchecked for authorization with sub-elements that follow.
- `<description>`: Describes the method permissions for an EJB (zero or more).
- `<role-name>`: Contains role name values that have been defined in a `<role-name>` element contained by the `<security-role>` elements defined previously (zero or more). Either zero or more `<role-name>` elements or an `<unchecked/>` element may be specified within the `<method-permission>` element, but not both.
- `<unchecked/>`: Is an empty element indicating that the associated method(s) are not checked for authorization. If it is indicated elsewhere in the deployment descriptor that a method is to be checked for authorization, the `<unchecked/>` element takes precedence and the method will not be checked. Either zero or more `<role-name>` elements or the `<unchecked/>` element may be specified within the `<method-permission>` element, but not both.
- `<method>`: Identifies particular EJB method(s) for which this access control specification applies (zero or more). The specification for a `<method>` element is identical to that defined for transactions described earlier in this chapter.

As an example, Listing 28.5 identifies three method access control specifications. The access control checking rules that apply to this specification for the `OrderManagerEJB` (mapped to the `OrderManagerEJBBean`) are defined here:

- All methods allow users in the `Managers` role access by default.
- The `create()` method, however, is not checked for authorization to invoke it.
- The `createNewOrder()` method is allowed access by those users in the `EveryUser` role in addition to those in the `Managers` role.

EJB Method Exclusion List (<exclude-list>)

There may be scenarios in which an assembler and a deployer determine that regardless of whether a method is defined in a home or object interface for access by clients, the method should simply not be invoked. Thus, regardless of whether the method uses

access control to determine whether a user in a particular role may invoke it, the optional `<exclude-list>` element defined within an `<assembly-descriptor>` element may identify such methods. The `<exclude-list>` element is defined here:

- `<exclude-list>`: Identifies those EJB methods that are not allowed to be invoked at all by EJB clients. If it is indicated elsewhere in the deployment descriptor that a method is to be checked or not checked for authorization, the `<exclude-list>` element takes precedence and the methods will not be invoked. The `<exclude-list>` element has the following sub-elements.
- `<description>`: Describes the method exclusion list for an EJB (zero or more).
- `<method>`: Identifies particular EJB method(s) for which this exclusion list specification applies (zero or more). The specification for a `<method>` element is identical to that defined for transactions described earlier in this chapter.

Listing 28.5 presented such an example in which the `getCustomer()` method defined for the `OrderManagerEJB` (mapped to the `OrderManagerEJBBean`) is not allowed to be accessed by clients of the bean.

Such declarative EJB security access control checking mechanisms provide a codeless way for determining whether a particular user in a particular role is allowed access to a particular method on a particular EJB. However, some enterprise applications may need to provide access control checking functionality in which access to a particular EJB method should be allowed based on some business-specific state and logic of the system associated with that user. For example, it may not simply be valid to say that all users belonging to a particular `employee` role have access or do not have access to particular getter and setter methods on some `TimeSheet` EJB that encapsulates employee timesheets in an enterprise human resource management application. Rather, you may want to implement a security checking mechanism that provides access control to such `TimeSheet` EJB getter and setter methods based on whether the identity of the invoking client matches some `employeeID` field state of the `TimeSheet` EJB. Although bean developers can use the standard programmatic EJB security access control checking mechanisms to implement the needed logic, some EJB container vendors may provide additional access control mechanisms to implement such features.

Standard EJB Principal Identification and Delegation

Thus far we've discussed the logical model involved with defining access control for EJBs. Such access control involved defining those methods on EJBs that can be accessed by users in particular roles. The role of a user is different from the identity defined by the user's security principal representation. As we saw, the principal identity of a user that is attempting to access an EJB can be gotten via a call to the `EJBContext` class's `getCallerPrincipal()` method. The identity used during access control decisions by the EJB container, however, can be different.

EJB Invocation Security Identity (<security-identity>)

A <security-identity> element defined within a <session>, <entity>, or <message-driven> bean element definition indicates what identity is used during access control decisions as explained here:

- <security-identity>: Defines the identity type to be used by the container when making access control decisions on the enclosed EJB with sub-elements that follow.
- <description>: Describes the security identification scheme for an EJB (zero or more).
- <use-caller-identity/>: Is an empty element to indicate that the principal identity of the user (that is, caller) should be used in access control decision making by the container. Either the <use-caller-identity/> element or a <run-as-identity> element is used inside of a <security-identity> element, but not both.
- <run-as-identity>: Contains zero or more <description> elements and a <role-name> element identifying that role which should be used in access control decision making by the container. Either the <use-caller-identity/> element or the <run-as-identity> element is used inside of a <security-identity> element, but not both.

Listing 28.5, in fact, illustrates the fact that any access control decisions made by the container for access to any methods on the OrderManagerEJB should use the caller's principal identity in those decisions. The caller identity must, of course, map to a valid role by the EJB container. As of EJB v2.1, this mapping is done in a vendor-specific fashion.

Alternatively, if we desired to deploy the OrderManagerEJB such that the container assumed that every user access attempt was made via the Managers role, we might have the following for the bean's <security-identity>:

```
<session>
  <display-name>OrderManagerEJB</display-name>
  <ejb-name>OrderManagerEJB</ejb-name>
  ...
  <security-identity>
    <description>
      Use Managers role for access control decisions.
    </description>
    <run-as>Managers</run-as>
  </security-identity>
</session>
```

Because no caller identity is associated with message-driven or TimedObject beans, the <use-caller-identity> element must never be specified for such bean types. Rather, the <run-as> identity must be defined or rely upon a container default value.

Such `<security-identity>` specifications are used only by the container when it is performing access control checking for an EJB. The principal identity returned from the `EJBContext.getCallerPrincipal()` method is that of the caller regardless of any `<run-as>` specification for the bean. Similarly, the `EJBContext.isCallerInRole()` method uses the caller identity versus any `<run-as>` security identity. If no authentication takes place, an anonymous user identity as defined by the container must be returned from `getCallerPrincipal()` and used in the `isCallerInRole()` decision making.

EJB Security Identity Delegation

Finally, if an EJB calls the services of another EJB, which calls the services of another EJB, and so on, then a means for principal identity delegation must be defined. The container defines this principal identity delegation in a vendor-specific fashion. However, the capability to allow principal identity to propagate as either the caller identity or the run-as identity value must be allowed by the container. This enables EJB developers to design their applications with such principal delegation schemes in mind.

Vendor-Specific EJB Access Controls

Although the roles specified by an assembler in the `<security-role>` elements of an `ejb-jar.xml` file define logical roles assumed by an EJB application, the container and EJB deployers must map these roles to actual user groups and/or users in the operational system. Additionally, the container and EJB deployers must manage how these roles relate to particular security domains from the operational system. Such mappings from logical security role to operational environment groups/users may be performed in an automated fashion by vendor tools without requiring the development of vendor-specific code.

As an example of a vendor-specific mapping tool, the BEA WebLogic Server comes equipped with a GUI administration console tool. The administration console can be used to map standard J2EE EJB defined role names to principal names that have meaning in an operational BEA WebLogic Server environment. The standard EJB security application roles that are stored in `<security-role>` elements for an EJB module must be mapped to valid groups used within a WebLogic security realm. The administration console then allows operational principal names to be mapped to the standard J2EE role names (exactly how such principal names are made available is described in the next section).

When performing such a mapping, the administration console populates a vendor-specific `weblogic-ejb-jar.xml` file. The `weblogic-ejb-jar.xml` file contains zero or more `<security-role-assignment>` elements which contain a standard EJB `<role-name>` element that maps to one or more WebLogic-specific `<principal-name>` elements. As an example included with this chapter's code, we specify that our standard `Managers` role name maps to an `ejavaManager` WebLogic username and that our standard `EveryUser` role name maps to an `ejavaUser`

WebLogic username in an operational WebLogic Server environment. The example mapping yields the following `weblogic-ejb-jar.xml` file snippet:

```
<weblogic-ejb-jar>
  ...
  <security-role-assignment>
    <role-name>
      Managers
    </role-name>
    <principal-name>
      ejavaManager
    </principal-name>
  </security-role-assignment>

  <security-role-assignment>
    <role-name>
      EveryUser
    </role-name>
    <principal-name>
      ejavaUser
    </principal-name>
  </security-role-assignment>
</weblogic-ejb-jar>
```

As another example, the J2EE reference implementation also provides similar tools to perform such mappings. The vendor-specific `sun-j2ee-ri.xml` file included with this chapter's example yields the following mapping:

```
<j2ee-ri-specific-information>
  <rolemapping>
    <role name="Managers">
      <principals>
        <principal>
          <name>ejavaManager</name>
        </principal>
      </principals>
      <groups>
        <group name="ejavagroup"/>
      </groups>
    </role>

    <role name="EveryUser">
      <principals>
        <principal>
          <name>ejavaUser</name>
        </principal>
      </principals>
```

```

    <groups>
      <group name="ejavagroup" />
    </groups>
  </role>
</rolemapping>
...
</j2ee-ri-specific-information>

```

Vendor-Specific EJB Authentication

Vendor-specific mappings from logical security role to operational environment groups/users may not require any vendor-specific code, but the exact means for how your container manages the access, addition, and removal of the operational groups/users within its auspices may indeed require vendor-specific code. That is, we saw the vendor-specific means for mapping standard EJB security roles to principal names in the preceding section, but how do we customize our vendor's server environment to be cognizant of such valid principal names? For example, if your enterprise manages principal information in a database, a vendor-specific means to access that information via custom JDBC calls may be required. However, you may also decide to use whatever means are provided by your particular vendor to automatically manage such information, which may or may not require specialized coding. Such methods might include a means to specify principal information in a simple text file, an XML file, or an LDAP structure.

Vendor-Specific Authentication Classes

Some vendors may use a simple text-based configuration file as the default means for storing usernames and passwords. User information may be added to such a file in the form of name/value entries as shown here:

```
security.password.userName=userPassword
```

Similarly, groups may be added to the file in the following form:

```
security.group.groupName=userName1, userName2, userName3, . . .
```

Configuring principal information using static configuration files, however, will be infeasible for most medium- to large-scale applications. Thus, many vendors will also provide a means to manage identities stored in alternative principal identification repositories (that is, security domains or realms). An alternative realm may be designated in a configuration file by setting a special property equal to a fully qualified class name representing the alternative realm. Such classes may then implement vendor-specific interfaces that enable the vendor's server environment to invoke operations on such class instances during operational processing of security-related events.

Realm types from a vendor might include means for authenticating users via information stored in a database, an LDAP server, or operating system-specific security directory services. Some vendors may also allow you to create a custom realm that enables

you to create a special realm class that implements methods to retrieve and modify user, group, and access control information. Thus, user profile information may be stored in a repository of your choice and managed in a business-specific fashion by these interface implementations. Although use of such vendor-specific authentication techniques may provide a lot of flexibility in how you implement operational user profiles, your operational environment-specific implementations of these interfaces will be based entirely on vendor-specific interfaces.

EJB Client Authentication with JNDI

A principal identity name and set of credentials must somehow propagate from EJB client to EJB server for authentication to occur. One way vendors may accomplish this is via setting of `javax.naming.Context.SECURITY_PRINCIPAL` and `javax.naming.Context.SECURITY_CREDENTIALS` properties onto a `javax.naming.InitialContext` during initial connectivity with the EJB server from the client. As an example for the BEA WebLogic Server, the `ejava.ejb.session.ordermgr.TestClient` EJB client wanting access to the previously secured `OrderManagerEJB` may pass username and credential information to the EJB server when obtaining an initial JNDI context. The `TestClient` is shown in Listing 28.6.

Listing 28.6 EJB Test Client (`TestClient.java`)

```
package ejava.ejb.session.ordermgr;
...
public class TestClient {
    private static String userName;
    private static String password;
    private static int queryCustomerID = 101;
    private static int queryOrderID = 1001;

    public static void printOrder(Order order) {
        System.out.println("ID : " + order.getOrderID());
        System.out.println("orderDate : " + order.getOrderDate());
        System.out.println(
            "shipInstruct : " + order.getShippingInstructions());
        System.out.println("shipDate : " + order.getShipDate());
        System.out.println("shipWeight : " + order.getShipWeight());
        System.out.println("paidDate : " + order.getPaidDate());
    }

    private static Integer createNewOrder(OrderManager orderManager)
        throws RemoteException {
        Order order = new Order();
        // Order is created today and paid today and shipped today.
        order.setOrderDate(new Date());
        order.setPaidDate(new Date());
    }
}
```

Listing 28.6 Continued

```

    order.setShippingInstructions("Leave at Door");
    order.setShipWeight(10.8);
    order.setShipCharge(32.54);
    order.setShipDate(new Date());
    // The customer info the order is being created for
    Customer customer = new Customer();
    customer.setCustomerID("" + queryCustomerID);
    order.setCustomer(customer);
    order.setBillingAddress(createAnAddress());
    order.setShippingAddress(createAnAddress());
    order.setItems(createItems());
    return orderManager.createNewOrder(order);
}

private static Address createAnAddress() {
    Address address = new Address();
    address.setAddress_1(" 2.1 EJB Way");
    address.setAddress_2(" 1");
    address.setCity(" J2EE ");
    address.setState("CA");
    address.setZipCode("14000");
    return address;
}

private static TShirt[] createItems() {
    TShirt itemOne = new TShirt();
    itemOne.setQuantity(2);
    itemOne.setTotalPrice(21.0);
    itemOne.setSize("XL");
    itemOne.setColor("White");
    itemOne.setUnitPrice(10.5);

    TShirt itemTwo = new TShirt();
    itemTwo.setQuantity(2);
    itemTwo.setTotalPrice(21.0);
    itemTwo.setSize("L");
    itemTwo.setColor("White");
    itemTwo.setUnitPrice(10.5);
    TShirt[] createdItems = { itemOne, itemTwo };
    return createdItems;
}

public static void main(String[] args) {
    try {
        // Create initial context

```

Listing 28.6 Continued

```

Context context = new InitialContext();
System.out.println(
    " looking for :"
    + EJBHelper.ORDER_MANAGER_HOME
    + " context:"
    + context);

// Look up order manager home reference
Object objectRef = context.lookup(EJBHelper.ORDER_MANAGER_HOME);
OrderManagerHome orderManagerHome =
    (OrderManagerHome) PortableRemoteObject.narrow(
        objectRef,
        ejava.ejb.session.ordermgr.OrderManagerHome.class);

// Create an order manager handle reference
OrderManager orderManager = orderManagerHome.create();

// Create a new order
Integer newOrderID = createNewOrder(orderManager);
System.out.println("New Order ID :" + newOrderID);

// Query for number of items in the order
int nItems = orderManager.numberOfItemsInAnOrder(queryOrderID);
System.out.println(" Number of Items in an Order :" + nItems);

// Query for orders based on customer ID
Vector orders = orderManager.queryOrders(queryCustomerID);
System.out.println("Received Orders :" + orders.getClass());
for (int i = 0; i < orders.size(); i++) {
    Order order = (Order) orders.elementAt(i);
    System.out.println("Received Order : " + i + ": Order Detail");
    printOrder(order);
}

// Get the average order value for a customer
float avgValue =
    orderManager.averageOrderValueOfACustomer(queryCustomerID);
System.out.println(
    " Average Order Value of a Customer :" + avgValue);

// Get a customer associated with an order
Customer customer
    = orderManager.getCustomer(newOrderID.intValue());
System.out.println(
    " Customer email address :" + customer.getEmail());

```

Listing 28.6 **Continued**

```

    } catch (Exception exception) {
        exception.printStackTrace();
    }
}
}
}

```

Note that the `TestClient` uses the parameterless form of the `InitialContext` constructor, which means that it is getting the properties used to construct an initial context from the system properties. The `runClient`, `runClientEveryUser`, and `runClientManager` scripts generated by the ANT scripts for these examples (that is, in the `examples/config/execscripts/ejava/ejb/security/basic` directory) pass a `CONTEXT_FACTORY` and `PROVIDER_URL` to the `TestClient` class's JVM as system properties. The `runClientEveryUser` and `runClientManager` example scripts additionally pass `CONTEXT_PRINCIPAL` and `CONTEXT_CREDENTIALS` properties to the JVM. The `CONTEXT_FACTORY` and `PROVIDER_URL` properties are established by the ANT script using the properties set in the example's `ejava/ejb/build.properties` file:

```

# JNDI Properties - when using WebLogic
jndi.factory=weblogic.jndi.WLInitialContextFactory
jndi.provider.url=t3://localhost:7001/

```

```

# JNDI Properties - when using the J2EE Reference Implementation
#jndi.factory=com.sun.enterprise.naming.SerialInitContextFactory
#jndi.provider.url=iiop://localhost:1050/

```

The `CONTEXT_PRINCIPAL` and `CONTEXT_CREDENTIALS` properties are ultimately read in from the example's `ejava/ejb/security/basic/build.properties` file as shown here:

```

# Everyuser principal
everyuser_role_principal=ejavaUser

# Everyuser credentials
everyuser_role_credentials=ejavaUser

# Manager role user principal
manager_role_principal=ejavaManager

# Manager role user credentials
manager_role_credentials=ejavaManager

```

Thus, when executing the respective `runClientXXX` scripts for this example, you should see the following behavior depending on which script is run:

- `runClient`: Runs with no username and password. Hence should be able to create an order manager, but that's all it has permissions to do.
- `runClientEveryUser`: Runs with the `everyuser.role.principal` username and `everyuser.role.credentials` password properties defined previously. Hence should be able to create an order manager and create a new order but nothing else.
- `runClientManager`: Runs with the `manager.role.principal` username and `manager.role.credentials` password properties defined previously. Hence should be able to invoke every operation on the order manager with the exception of the `getCustomer()` method because that is in the exclude list for the bean.

On the EJB server side, if a vendor-specific realm is being used, special callbacks onto the realm implementation class are made to pass such principal and credential information to enable your code to provide authentication of such users with the system. A vendor-specific object returned within the vendor's container representing an authenticated user of the system is then associated with a server-side thread handler for the client. When subsequent requests are made from that same client associated with the thread, the client's credentials from its thread handler then can be used in the process of providing access control for EJBs.

Thus, if the EJB client that was previously authenticated during an initial JNDI context creation operation can access the system, it may attempt to look up EJB home references and invoke operations on those EJBs as usual. If an EJB's methods have been restricted using the standard and vendor-specific access control mechanisms defined previously, the security-critical EJB operations will be checked by the server during runtime operation. Running the `TestClient` of Listing 28.6 illustrates this behavior.

EJB Connection Security

The EJB client authentication sequence defined in the preceding section passed user credential information into a JNDI context object with no guarantee about the secured nature of the client-to-server connection. Thus, the client's credential information could very well have been transferred to the server in the clear. For certain types of EJB clients and application scenarios, this may be acceptable. For example, if the EJB client is a Java Servlet or JavaServer Page, and the Web server sits behind the same trusted computing base and firewall as the EJB application server, this may be acceptable for certain applications. However, for other enterprise applications, the connection between EJB client and EJB server needs to operate over some secured socket mechanism.

The means for securing the connection between EJB client and server is described in the EJB v2.1 specification such that interoperability may be assured. EJB uses IIOP for communications and secure communications interoperability is defined within the OMG's Common Security Interoperability v2 specification. EJB servers must adhere to conformance level 0 of that specification to ensure interoperability of security context information passed between client and server, as well as between servers.

One basic implication of this specification on an EJB server is that SSL v3.0 and TLS v1.0 are required as encryption standards for transport-level security. X.509 certificates, Kerberos, and a set of standard cipher suites are all delineated as standards within this encryption security umbrella. Furthermore, the SSL host and port information of a server is required to be baked into the home or remote EJB interface stub IOR so that the client knows how to connect with a server via SSL. If authentication information isn't passed around at this secured transport level, such information is passed inside of the IIOP messages themselves.

EJB Principal Delegation

The default means for propagating principal identity should allow principal identity to propagate from one EJB to another EJB. Thus, in a chain of calls made from one EJB to another, the principal identity associated with the original EJB client that initiated the call sequence will be propagated. Exactly how principal identity is propagated from the initial EJB client to the EJB server is left up to the vendor. An EJB vendor must provide a means during deployment for selecting which identity propagation policy should be employed. As a minimum requirement, the EJB client's principal identity used to connect to the EJB server will be able to be used as the principal identity to be delegated to other EJBs within the EJB server environment. The run-as identity may also be used and propagated across an EJB call chain.

EJB Security Auditing

EJB server vendors may also provide a means for auditing security-critical events. Security-critical events may include logging the generation of any security-related exceptions, successful and failed authentication attempts, and failed EJB access attempts. Vendors, for example, may enable you to implement a vendor-specific auditing class. A special security auditing configuration property in the vendor's specific configuration files might then be set to this class name. The vendor's server may then invoke methods on this class when an attempted authentication occurs, an authorization request is made, or an invalid user certificate is propagated to the server.

J2EE and EJB Security with JAAS

In the discussion thus far, an underlying vendor-specific means to authenticate a user was assumed to be used on the EJB server side. Although such vendor-specific means can be employed with minimal effects to your code base, a core principle of J2EE is to enable you to build enterprise applications that are portable across servers. A reliance on a vendor-specific authentication mechanism or custom realm will somewhat tie you to that vendor's EJB container.

The EJB v2.1 and J2EE v1.4 specifications do not explicitly require a vendor to support the standard security and authentication offerings for Java. Chapter 17 briefly introduced the Java Authentication and Authorization Service (JAAS) as a means for limiting

access to security-critical resources based on an authenticated user identity. With JAAS, APIs for login and logout provide a standard technique for authenticating users and passing around secure context and credential information. Different underlying authentication and authorization models can be plugged into the JAAS service provider interface model while enabling API users to have a stable and standard interface. Although JAAS is now included with the J2SE v1.4, there is no explicit requirement in the EJB and J2EE specifications that EJB containers must use JAAS to provide server-side authentication services.

Nevertheless, many vendors are slowly embracing JAAS for server-side EJB authentication, and a future specification may require more wholehearted support of JAAS if it becomes a popular means for providing such services. This section discusses JAAS in detail and specifically describes how it may be used in EJB environments. We take the approach in this section of first describing the basic mechanisms of JAAS for authentication and authorization so that you can best understand and focus on the JAAS APIs and how they may be used in various J2EE container environments. We then discuss how JAAS may be integrated with EJB environments so that you can understand how JAAS would be most commonly used and how it can offer you a vendor-independent authentication and authorization approach.

JAAS Subjects

The core abstractions involved with JAAS that are common to both authentication and authorization are depicted in Figure 28.6. An abstraction for a subject must exist to enable both the authentication of a subject with the system and the authorization of a subject's access to valued resources. Because a subject can have more than one principal name, a subject abstraction must also encapsulate a collection of principal objects. Furthermore, a subject can also have one or more public and private credentials to present to a system for authentication. Credential objects may be of any form and can implement special interfaces that signify the fact that they can be refreshed for an extended validity period or signify that they can be destroyed from the system.

The `javax.security.auth.Subject` class is a `Serializable` and `final` class used to encapsulate an entity that wants to utilize some security-critical aspect of a system. Such an entity may correspond to a person, another system, or perhaps a group of people or systems. Because subjects can have multiple ways to identify themselves as a unit, the `Subject` class maintains a collection of one or more `java.security.Principal` objects in a standard `java.util.Set` collection. Thus, for example, if the `Subject` does correspond to a person, `Principal` objects may be associated with the `Subject` for that person's name, employee identification number, Social Security number, and so on.

`Subject` objects may also be associated with credentials that define information used to validate such principal identities. Credential objects associated with a `Subject` are also stored in `java.util.Set` collections but have no required type except for being basic `java.lang.Object` types. Credentials may be either public or private and thus are managed in different collections. Public credentials, such as public certificates, can be

shared with other users of a system, whereas private credentials, such as private keys, should not be made accessible to anyone but the owning subject.

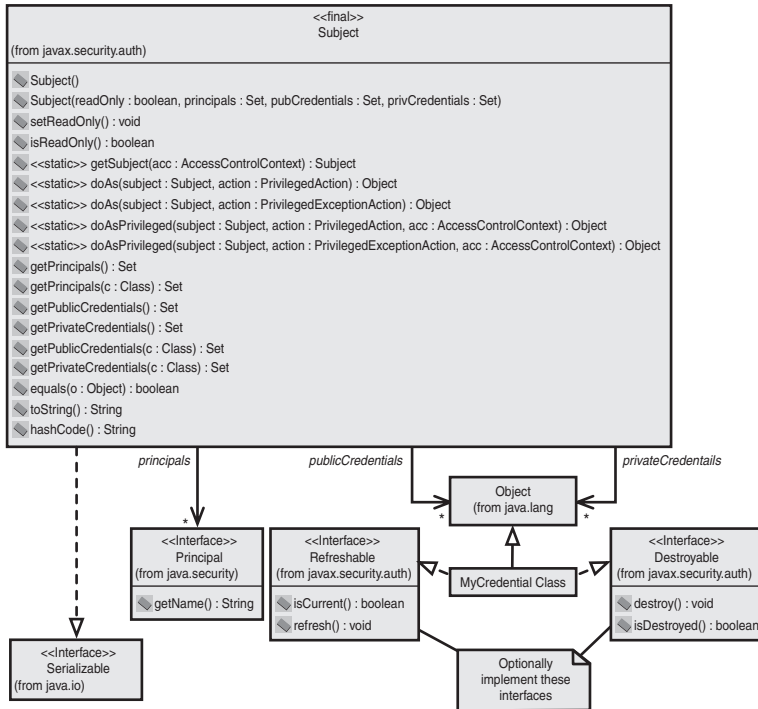


Figure 28.6 Core JAAS abstractions.

Creating Subjects

Although a default `Subject` constructor is provided, the `Subject(boolean, Set, Set, Set)` constructor is used primarily as the means to create a subject with a set of principals, set of public credentials, and a set of private credentials. A `boolean` value is also passed into this constructor to indicate whether the `Subject` object is read-only. As an example, we might have this:

```
// Create a principal and credentials hash sets
HashSet principals = new HashSet();
HashSet privateCredentials = new HashSet();
HashSet publicCredentials = new HashSet();

// Add initial elements to the sets
principals.add(new MyPrincipal("edison"));
publicCredentials.add("publicPassword");
```

```
privateCredentials.add("privatePassword");

// Create a Subject
Subject subject = new Subject(false, principals,
                             publicCredentials, privateCredentials);
```

Manipulating Subject Attributes

After a handle to a Subject object is either created or received, the principal objects and credential objects may be retrieved. The `getPrincipals()`, `getPublicCredentials()`, and `getPrivateCredentials()` methods all return a Set object containing the associated object types. Any updates performed on the Set objects from a non-read-only Subject will result in the Set object associated with the Subject being updated as well. Here's an example:

```
// Retrieve the public credential
String publicCredential =
    (String) ((subject.getPublicCredentials()).iterator()).next();
System.out.println("Public credential = " + publicCredential);

// Retrieve the private credential
String privateCredential =
    (String) ((subject.getPrivateCredentials()).iterator()).next();
System.out.println("Private credential = " + privateCredential);

// Retrieve a principal and display its value
// and number of principals in Subject (currently has one)
MyPrincipal principal =
    (MyPrincipal) ((subject.getPrincipals()).iterator()).next();
System.out.println("Principal object in Subject = " + principal.getName());
System.out.println("Number of Principal objects in Subject = "
    + subject.getPrincipals().size());

// Modify the principal (is OK)
principal.setName("watson");
// Modify the set (is also OK)
try{
    subject.getPrincipals().add(new MyPrincipal("joe"));
}
catch(IllegalStateException ex){
    System.out.println("This statement won't be called");
    ex.printStackTrace();
}

// See number of principals in Subject (now has two)
System.out.println("Number of Principal objects in Subject = "
    + subject.getPrincipals().size());
```

Alternatively, the `getPrincipals(Class)`, `getPublicCredentials(Class)`, and `getPrivateCredentials(Class)` methods all return a `Set` object containing the associated object types whose class type is a subclass or is the type of class specified by the `Class` input parameter. Because these method calls return copies of the `Set` objects that they return, updates made to the `Set` objects do not affect the state of the `Subject` object. However, any updates made to the individual elements within the `Set` object will affect the `Subject` object state. For example:

```
// Modify the set (only modifies the copy and not the Subject)
try{
    Set mySetCopy = subject.getPrincipals(MyPrincipal.class);
    mySetCopy.add(new MyPrincipal("bart"));
}
catch(IllegalStateException ex){
    System.out.println("This statement won't be called");
    ex.printStackTrace();
}

// See number of principals in Subject (still has two)
System.out.println("Number of Principal objects in Subject = "
    + subject.getPrincipals().size());
```

If a `Subject` is set to be read-only, then attempted modifications to the `Subject` object's principal `Set`, public credentials `Set`, and private credentials `Set` will not be permitted and will result in the throwing of an `IllegalStateException`. The read-only nature of the `Subject` can be determined from `Subject.isReadOnly()`. The `Subject.setReadOnly()` can be used to set a `Subject` to be read-only, but there is no way to make a `Subject` writable after it has been set to read-only. Here's an example:

```
// Set the Subject to read-only
subject.setReadOnly();

// Modify the principal value (is still OK)
principal.setName("sam");
// Modify the set (is NOT OK and will throw IllegalStateException)
try{
    subject.getPrincipals().add(new MyPrincipal("marcus"));
}
catch(IllegalStateException ex2){
    System.out.println("This statement WILL be called");
    ex2.printStackTrace();
}
```

The `Subject` class also defines a `getSubject()` method, two `doAs()` methods, and two `doAsPrivileged()` methods. These methods involve authorization concepts and are covered later in this section.

Specializing Subject Credentials

As a final note, two interfaces, `javax.security.auth.Refreshable` and `javax.security.auth.Destroyable`, may also be used with credential objects associated with a `Subject`. If a particular credential class implements the `Refreshable` interface, such a class designates that the credential can have its validity period extended (that is, refreshed). If a credential class implements the `Destroyable` interface, the credential information contained by that class can be wiped from memory.

Authentication with JAAS

Authentication in JAAS involves a client application that communicates with a login context to present any principal credential information to JAAS, as well as perform login and logout operations. The login context communicates with a modular login module service provider interface to interact with any underlying authentication-specific technologies. Such an architecture follows the pattern employed by the Pluggable Authentication Module (PAM) framework for shielding applications from the underlying authentication technologies. Login modules used by an application are configured using a special configuration abstraction that determines how to load and initialize the login modules. Figure 28.7 depicts the basic set of JAAS abstractions involved with authentication.

Login Context Construction

The `javax.security.auth.login.LoginContext` class is the primary class utilized by clients to the JAAS API. A `LoginContext` provides the interfaces necessary for subjects to log in and log out of the system. The `LoginContext` class shields JAAS clients from the underlying authentication mechanisms.

Clients first create an instance of a `LoginContext` object using one of four constructor options. The `LoginContext(String)` constructor is used to create a `LoginContext` for a particular named authentication configuration. The authentication configuration name is simply used to identify particular methodologies for logging into the application and particular parameters associated with those methodologies. All `LoginContext()` constructors require an authentication configuration name to be supplied. Because no `Subject` is associated with the `LoginContext(String)` constructor call, the `LoginContext` will create one for the user and expects to receive principal and credential information via an alternative mechanism. The `LoginContext(String, Subject)` method creates a handle to a named authentication configuration with a particular `Subject` to be authenticated. As an example, we might have this:

```
// Get our dummy Subject ("edison")
Subject subject = ...

// Create LoginContext
LoginContext loginContext = null;
```

```

try{
    loginContext = new LoginContext("PasswordExample", subject);
}
catch(LoginException loginException){
    loginException.printStackTrace();
    System.exit(1);
}

```

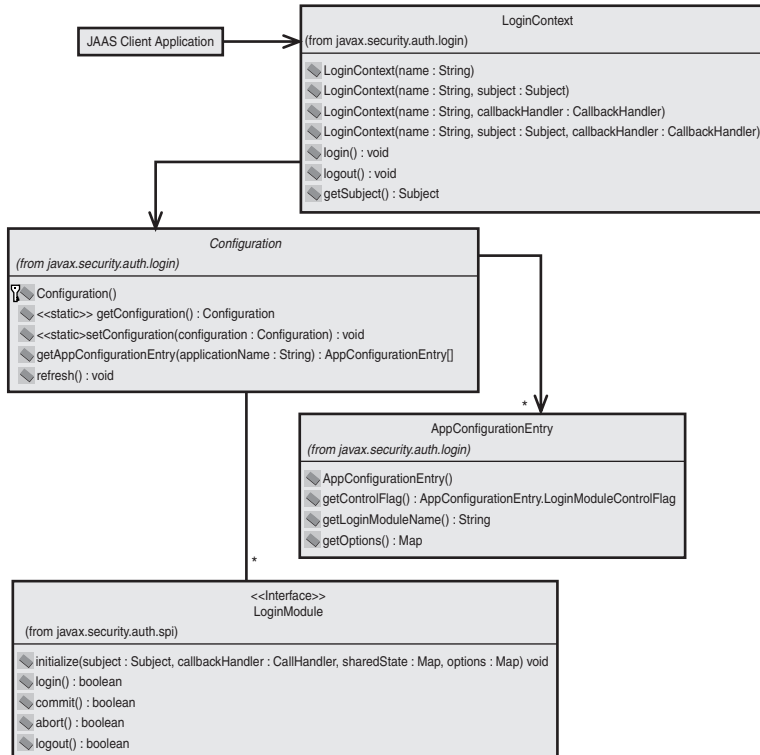


Figure 28.7 JAAS authentication abstractions.

Login Module Configuration

During construction time, the `LoginContext` object consults a `javax.security.auth.login.Configuration` object to load the login modules associated with the application. The `Configuration` object determines which login modules should be loaded and the order in which they should be loaded. Because

`Configuration` is an abstract class, a concrete subclass must be specified somewhere for your JAAS applications to run properly. To override the default `Configuration` implementation, you must alter your J2SE `<JAVA_HOME>/jre/lib/security/java.security` file such that a `login.configuration.provider` property points to a fully qualified `Configuration` subclass name. Otherwise, the default `Configuration` implementation class assumes that a special file for configuring login modules is defined according to the following form:

```
AuthenticationConfigurationName
{
  ModuleName AuthenticationFlag ModuleOptions;
  ModuleName AuthenticationFlag ModuleOptions;
  ...
};

AuthenticationConfigurationName
{
  ModuleName AuthenticationFlag ModuleOptions;
  ModuleName AuthenticationFlag ModuleOptions;
  ...
};

...

other
{
  ModuleName AuthenticationFlag ModuleOptions;
  ModuleName AuthenticationFlag ModuleOptions;
  ...
};
```

Each *AuthenticationConfigurationName* identifies one or more login modules that should be used to authenticate users for the application that desires to use this particular configuration. Each login module entry corresponds to a class that implements the `javax.security.auth.spi.LoginModule` interface. The `ModuleName` entry thus relates to a fully qualified `LoginModule` implementation class name. The default `Configuration` class implementation will proceed down the list for the *AuthenticationConfigurationName* and attempt any authentication with a `LoginModule` based on its ordered entry in the list. If no *AuthenticationConfigurationName* is found for the application, the default other application configuration will be used.

Each login module entry also defines an *AuthenticationFlag* value that defines how the authentication proceeds with the login modules configured for this

authentication configuration. The basic values that can be specified for the *AuthenticationFlag* are listed here:

- **Required:** The associated login module must succeed with authenticating the subject for authentication to be considered successful. The authentication process still continues down the list for other login modules, however.
- **Requisite:** The associated login module must succeed for authentication to be considered successful. The authentication process terminates as failed if a failed login occurs.
- **Sufficient:** The associated login module's successful authentication is sufficient and the authentication may terminate as successful if a successful login occurs. If a **Sufficient** flag is configured, the **Required** and **Requisite** modules preceding this module only must have succeeded.
- **Optional:** The associated login module's capability to authenticate the subject is not required. The authentication process still continues down the list for other login modules. At least one **Optional** or one **Sufficient** module must have succeeded if no **Required** or **Requisite** modules are configured.

Finally, the *ModuleOptions* entry per login module is used to define any configuration parameters that can be passed to the associated `LoginModule` implementation during initialization. Such parameters are defined as a series of *name=value* pairs.

As an example, we might define an example `jaas.config` file for an example application as shown here:

```

PasswordExample
{
    ejava.jaas.AuthenticationExampleModule Required
    ↪ fileName=credentials.properties;
};

```

Thus, our call to `new LoginContext("PasswordExample", subject)` as illustrated earlier will induce the default `Configuration` implementation to configure our application to make use of the `PasswordExample` authentication configuration information defined in the example `jaas.config` file. A special `ejava.jaas.AuthenticationExampleModule` class (to be defined soon) serves as an example required login module implementation to be used for authenticating subjects. Note that the example `AuthenticationExampleModule` class also utilizes a special `fileName` module option during its initialization process.

Login Module Configuration File Location

The location of the `jaas.config` file can be specified for our application in one of three ways. From the command line, we can set a special system property named `java.security.auth.login.config` to the location of our authentication configu-

ration file. Thus, if the `jaas.config` file is in our current directory, we might execute the example like this:

```
java -Djava.security.auth.login.config=jaas.config
➔ ejava.jaas.AuthenticationExample
```

We can also define a series of `login.config.url.n` properties in the `<JAVA_HOME>/jre/lib/security/java.security` file to point to the location of authentication configuration files. The order in which configuration files are loaded is implied by the numeric value associated with the `n` in `login.config.url.n`. For example:

```
login.config.url.1=file:C:/ejava/jaas/certificate.config
login.config.url.2=file:C:/myb2c/config/b2c.config
login.config.url.3=file:C:/myb2b/config/b2b.config
```

If no location is specified via the system property or from the `java.security` file, JAAS will attempt to load configuration information from the default `<USER_HOME>/ .java.login.config` file.

Login Module Initialization

After loading and instantiating any objects that implement the `LoginModule` interface, the `Configuration` class invokes the `initialize()` method on each `LoginModule` object. The `initialize()` method takes as input parameters the `Subject` to be authenticated, any `CallbackHandler` object for interacting with the end user (to be described shortly), a `Map` of any data shared with other `LoginModule` objects, and a `Map` of module-specific options for this `LoginModule` (such as those read from the `ModuleOptions` entry described earlier). Although application developers most often utilize off-the-shelf `LoginModule` implementations, such as those defined at <http://java.sun.com/products/jaas/>, we demonstrate a simple `LoginModule` here to better illustrate the authentication process involved with JAAS. The example `ejava.jaas.AuthenticationExampleModule` class implements a simple `LoginModule` that we will use to authenticate users based on credential information stored in a properties file.

The `AuthenticationExampleModule.initialize()` method stores some basic information in addition to the name of the credential properties file as shown here:

```
package ejava.jaas;
...
public class AuthenticationExampleModule implements LoginModule
{
    private Subject subject = null;
    private String credentialFileName = null;
    private CallbackHandler callbackHandler = null;
    private Map otherState = null;
    private boolean loginSucceeded = false;
```

```

public void initialize(Subject aSubject,
                      CallbackHandler aCallbackHandler,
                      Map sharedState, Map options) {

    // Get Subject, callback handler, and shared state
    subject = aSubject;
    callbackHandler = aCallbackHandler;
    otherState = sharedState;

    // Get filename of credential file
    credentialFileName = (String) options.get("fileName");

    // Print out info
    System.out.println("Start AuthenticationExampleModule with file "
                      + credentialFileName);
}
...
}

```

The Authentication Process

After each `LoginModule` is initialized, user authentication may proceed. A JAAS client will induce the authentication process to begin by invoking the `LoginContext.login()` method. The `LoginContext.login()` method will in turn invoke the `login()` method on each `LoginModule` object configured for this application. The exact nature of the `login()` implementation is a function of the particular authentication technique to be employed by the `LoginModule` implementation. During this first phase of authentication, our `AuthenticationExampleModule` implementation reads credential information from a file and then simply compares it with the established credential information passed in with the `Subject` as shown here:

```

public boolean login() throws LoginException
{
    // First dynamically load credentials file
    Properties credentialsInfo = this.loadCredentialsFile();

    // Get principal list
    Iterator principals = (subject.getPrincipals()).iterator();

    // For each principal, attempt login
    while(principals.hasNext()){
        // Get principal and print out info
        Principal principal = (Principal) principals.next();
        String name = principal.getName();
        System.out.println("Attempting login for " + name);
    }
}

```

```

// If principal is in the credentials file...
if(credentialsInfo.containsKey(name)){
    // Get and print out password (since is just a test program)
    String password = (String) credentialsInfo.get(name);
    System.out.println(name + " password is " + password);
    Set privateCreds = subject.getPrivateCredentials();
    if(privateCreds.contains(password)){
        System.out.println("Success in login");
        loginSucceeded = true;
        return true;
    }
}
else{
    System.out.println("Fail in login");
    loginSucceeded = false;
    throw new LoginException("Not a valid subject");
}
}

System.out.println("Fail in login");
loginSucceeded = false;
return false;
}

private Properties loadCredentialsFile() throws LoginException
{
    // Throw exception if no credentials file
    if(credentialFileName == null){
        System.out.println("Fail in login");
        loginSucceeded = false;
        throw new LoginException("No credentials file");
    }

    // Load the properties
    Properties properties = new Properties();
    try{
        FileInputStream fin = new FileInputStream(credentialFileName);
        properties.load(fin);
    }
    catch(Exception e){
        System.out.println("Fail in login");
        loginSucceeded = false;
        throw new LoginException("No credentials file");
    }

    return properties;
}

```

If the subject was successfully authenticated during the overall authentication process, a second phase of the authentication process results in the `commit()` method being invoked on each `LoginModule` implementation object. If a particular `LoginModule` successfully authenticated the subject, it should associate any `Principal` and credential objects with the `Subject` contained by this `LoginModule`. For our `AuthenticationExampleModule`, we assume that a populated `Subject` object will be associated with the `LoginModule` during initialization and thus there is no need to re-associate any `Principal` or credential information with the `Subject`. When a `LoginContext` constructor is called with no `Subject`, the application logic of the invoked `LoginModule` implementations will have to retrieve the principal and credential information from a user via some other mechanism (such as callbacks) and associate this information with a `Subject` during the call to `LoginModule.commit()`. If the authentication for this `LoginModule` failed, the `LoginModule` implementation must destroy any saved state. A value of `true` is returned from `commit()` if the method succeeded.

If the subject was not successfully authenticated during the overall authentication process, the second phase of the authentication process results in the `abort()` method being invoked on each `LoginModule` implementation object. This method implementation must destroy any saved state. A value of `true` is returned from `abort()` if the method succeeded.

The JAAS client application may invoke `getSubject()` on the `LoginContext` object anytime after authentication succeeded and obtain a handle to the authenticated `Subject` object. If authentication failed, the `getSubject()` method returns `null`.

Finally, when the JAAS client application invokes `logout()` on a `LoginContext` object, the `LoginContext` object will invoke `logout()` on each `LoginModule` configured for this application. Each `LoginModule` should destroy any `Principal` and credential state stored. A value of `true` is returned from `LoginModule.logout()` if the method succeeded.

Callback Handling

Callbacks provide a means for JAAS application clients to be notified of certain events during the authentication process and to provide data to the authentication process. Figure 28.8 depicts the basic structure for implementing callbacks in JAAS. A special `javax.security.auth.callback.CallbackHandler` interface is implemented by JAAS applications that want to be consulted during the authentication process. A `CallbackHandler.handle()` method is implemented by JAAS application clients to receive a collection of `javax.security.auth.callback.Callback` objects when a `LoginModule` is performing the login process. Such callback objects contain information relevant to the particular type of authentication being performed. JAAS application clients then retrieve and provide any necessary information via `Callback` objects for authentication to proceed.

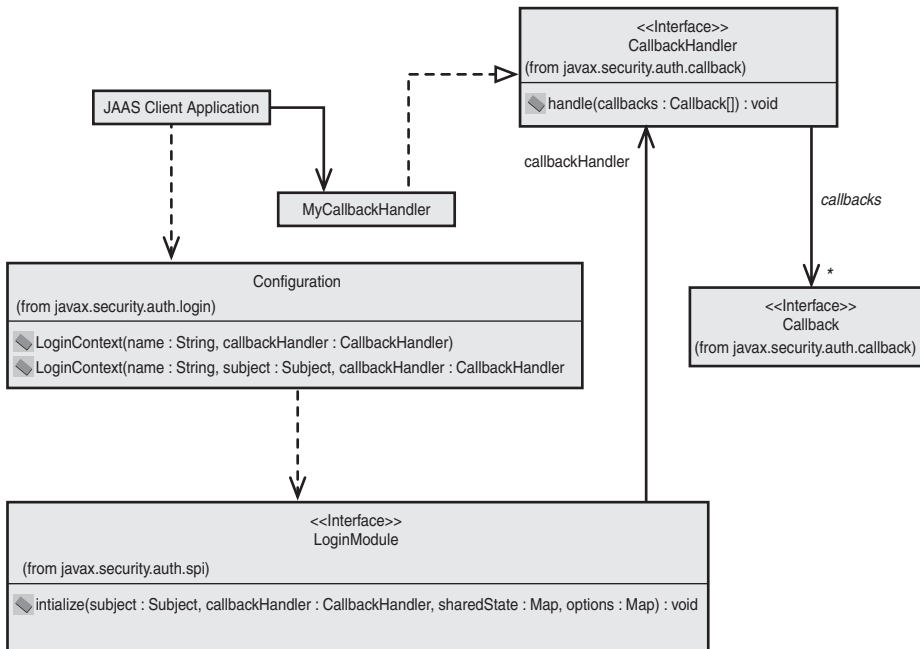


Figure 28.8 Basic JAAS callback handling.

CallbackHandler objects are registered with a LoginContext via one of two constructors shown in Figure 28.8. LoginModule implementations may save such CallbackHandler references for later use when a call to LoginModule.initialize() is made. A LoginModule may then invoke the services of the CallbackHandler object during the authentication process via a call to CallbackHandler.handle(). The actual Callback objects passed into the handle() method is a function of the particular type of authentication being performed. Figure 28.9 depicts the basic collection of Callback implementations equipped with JAAS included with the J2SE v1.4.

Although we will not go into a detailed description of each Callback mechanism here, we should point out the fact that the basic callback types shown in Figure 28.9 have the following general roles:

- **ChoiceCallback**: Used to retrieve a list of choices available during authentication and enable the client application to select particular authentication options.
- **ConfirmationCallback**: Used to retrieve authentication confirmation information and for clients to establish particular confirmation information to drive the authentication process.
- **LanguageCallback**: Used to get and set geographic and regional locale information during authentication.

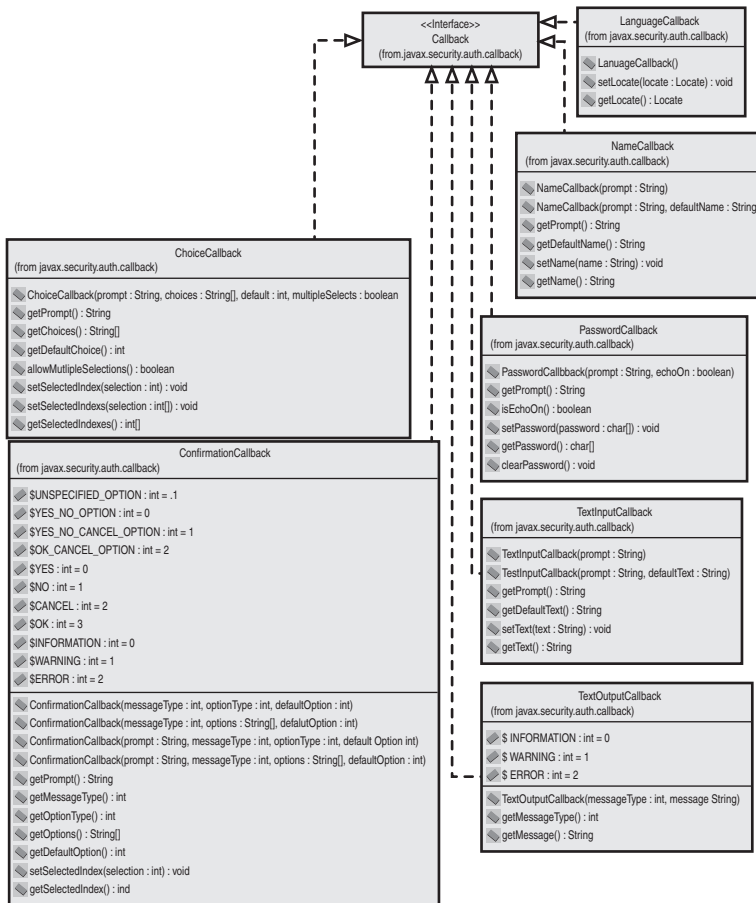


Figure 28.9 Standard JAAS callback implementations.

- **NameCallback:** Used to get and set any username information used during authentication.
- **PasswordCallback:** Used to get and set any user password information used during authentication.
- **TextInputCallback:** Used to provide any general text input to the authentication process.
- **TextOutputCallback:** Used to receive any general textual output messages from the authentication process.

As a general example, if a `LoginModule` wants to receive username and password information from a user during authentication, we might have the following inside of a `login()` method implementation of a `LoginModule`:

```
// If have no callback handler...cannot login
if (callbackHandler == null){
    throw new LoginException("No handler available");
}

// Create a username and password callback
NameCallback nameCallback = new NameCallback(" Enter Username ");
PasswordCallback passwordCallback =
    new PasswordCallback(" Enter Password ", false);

// Create the callback array for the CallbackHandler
Callback callbacks[] = new Callback[2];
callbacks[0] = nameCallback;
callbacks[1] = passwordCallback;

// Invoke handler() on the CallbackHandler
try{
    callbackHandler.handle(callbacks);
}
catch(Exception ex){
    ex.printStackTrace();
    loginSucceeded = false;
    throw new LoginException(" Cannot login.");
}

// Now should have the username and password
String userName = nameCallback.getName();
char [] password = passwordCallback.getPassword();
```

On the JAAS client application side, the handling of this callback will be application specific. That is, the username and password may be obtained from a GUI, via some distributed object call, or via some other mechanism. However, a basic skeleton for such a `handle()` implementation might follow as shown here:

```
public void handle(Callback[] callbacks)
    throws UnsupportedCallbackException, IOException
{
    // Get callbacks
    NameCallback nameCallback = (NameCallback) callbacks[0];
    PasswordCallback passwordCallback = (PasswordCallback) callbacks[1];
```

```

    // Get username prompt
    String prompt = nameCallback.getPrompt();
    // Get name from user (ap-specific)...
    String userName = ...
    // Now set name
    nameCallback.setName(userName);

    // Get password prompt
    passwordCallback.getPrompt();
    // Get password from user (ap-specific)...
    char [] password = ...
    // Now set password
    passwordCallback.setPassword(password);

    ...
}

```

Authorization with JAAS

After successfully authenticating a subject, the authorization features unique to JAAS may be employed to limit access to valued resources based on the authenticated subject information. By virtue of incorporating JAAS within the J2SE, the JAAS authorization model resulted in the extension of the Java 2 security model to include additional access control decision-making logic based on the subject requesting access.

The `Subject.doAs()` and `Subject.doAsPrivileged()` methods may be used to perform security-critical actions as a particular `Subject`. These static methods (as shown in Figure 28.6) have a role similar to the `AccessController.doPrivileged()` methods. However, these methods also require an additional `Subject` object input parameter. Fine-grained access control for such `Subject.doXXX()` calls are defined in the Java security policy file.

Java Security Policies with Subjects

In the default implementation of Java security policy management, a security policy is defined in an ASCII text file. Recall from Chapter 17 that the Java security policy file has the following general form:

```

grant codebase "URL", SignedBy "list of names",
    Principal [principal_class_name] "principal_name",
    Principal ...
{
    permission permission_class_name [ "target name" ] [, "actions"]
        [, SignedBy "list of names"];
    permission ...
};

```

As discussed in Chapter 17, the basic policy file includes the specification of a security domain beginning with the `grant` keyword and contains one or more permission definitions for the particular domain of protection. Each permission entry begins with the `permission` keyword and is followed by the fully qualified permission `class_name`. Permission target names and a comma-separated list of actions also follow a permission designation. The `SignedBy` field following the permission is provided with a comma-separated list of alias names that indicate who must have signed the Java code for the permissions class. Each grant entry delimiting a domain of protection can also contain a `SignedBy` field designating the fact that code being executed in the JVM associated with the grant entry must be signed by the named signers. A `CodeBase` field may also be associated with a grant entry to relate a set of permissions with a resource location from where the code is loaded.

Java security policy files may also specify one or more principal identification values associated with each grant entry. A fully qualified principal class name identifies the type of `Principal` object associated with valid principals to which the associated permissions apply. The actual principal name must also be supplied. Thus, any `Subject` to which the associated permissions apply must have all the specified principal values associated with the Java security policy entry.

We thus define the following simple entry in a Java security file for the example `AuthenticationExample` application:

```
grant codebase "file:authaction.jar",
    Principal ejava.jaas.MyPrincipal "edison"
{
    permission java.io.FilePermission "myLogFile.txt", "write";
};
```

Thus, our `AuthenticationExample` will only allow write permissions to a `myLogFile.txt` file for an authenticated subject containing the principal name of `edison`. Furthermore, the codebase for this permission is associated with code loaded from an `authaction.jar` file.

Performing Security-Critical Actions

The security-critical action to be performed by our example application attempts to write to a file named `myLogFile.txt`. We will encapsulate this security-critical operation within a `PrivilegedAction` class as can be done with other standard Java 2 security-critical actions. We then compile and insert this security-critical action class into the `authaction.jar` file as referenced by our Java security policy file described previously. The definition of this `PrivilegedAction` class is rather simple for our demonstration purposes and is encapsulated with an `AuthorizationAction` class as shown here:

```
package ejava.jaas;

import java.security.PrivilegedAction;
import java.io.*;
```

```

public class AuthorizationAction implements PrivilegedAction
{
    public Object run()
    {
        try{
            FileOutputStream fout = new FileOutputStream("myLogFile.txt");
            DataOutputStream dout = new DataOutputStream(fout);
            dout.writeChars("Attempted log!");
        }
        catch(Exception e){
            e.printStackTrace();
        }

        return null;
    }
}

```

The invocation of this privileged action by our `AuthenticationExample` occurs after our subject has been authenticated by the system. Thus, we might simply attempt to perform this security-critical action as shown here:

```

// Get our subject after authentication
Subject mySubject = loginContext.getSubject();

// Now perform the security-critical authorization action
Subject.doAs(mySubject, new AuthorizationAction());

```

JAAS Security Authorization Permissions

Upon integration of JAAS with the J2SE, the `java.security.ProtectionDomain` class was extended to define a `ProtectionDomain(CodeSource, PermissionCollection, ClassLoader, Principal[])` constructor. This allows a `ProtectionDomain` to be defined to associated with one or more principals as required by subject-oriented authorization to incorporate subjects into a Java security policy grant entry. The `java.security.Policy` class was also extended to define a `getPermissions(ProtectionDomain)` method that returns a `PermissionCollection` object associated with a particular `ProtectionDomain` which could include subjects associated with a Java security policy grant entry.

A few permissions are also defined in the `javax.security.auth` package to define permissions related to JAAS-oriented authentication. The `javax.security.AuthPermission` class encapsulates authentication permissions that have a target name (but no action lists) that can be used to grant permissions for access to `Subject`, `LoginContext`, and `Configuration` objects. The target names of an `AuthPermission` may be one of the following:

- `doAs`: Allows for the invocation of `Subject.doAs()` methods.
- `doAsPrivileged`: Allows for the invocation of `Subject.doAsPrivileged()` methods.

- `getSubject`: Allows for the invocation of `Subject.getSubject()` methods to obtain a handle to a `Subject` object for the current thread.
- `getSubjectFromDomainCombiner`: Allows for the invocation of the `SubjectDomainCombiner.getSubject()` method. The `javax.security.auth.SubjectDomainCombiner` class extends the `java.security.DomainCombiner` class to update protection domains with subject permissions defined with the Java security policy.
- `setReadOnly`: Allows for permission to set the `Subject` to be read-only.
- `modifyPrincipals`: Allows for permission to modify a `Set` containing a `Subject` object's principals.
- `modifyPublicCredentials`: Allows for permission to modify a `Set` containing a `Subject` object's public credentials.
- `modifyPrivateCredentials`: Allows for permission to modify a `Set` containing a `Subject` object's private credentials.
- `refreshCredential`: Allows for permission to invoke the `Refreshable.refresh()` method in order to refresh a credential object's state.
- `destroyCredential`: Allows for permission to invoke the `Destroyable.destroy()` method to destroy a credential object's state.
- `CreateLoginContext.{name}`: Allows for permission to create a `LoginContext` object with the specified name or `*` for any name.
- `getLoginConfiguration`: Allows for permission to retrieve the login Configuration.
- `setLoginConfiguration`: Allows for permission to set the login Configuration.
- `refreshLoginConfiguration`: Allows for permission to refresh the login Configuration.

The `javax.security.auth.PrivateCredentialPermission` class is used to grant or limit access to private credential objects that are associated with a `Subject`. The basic format for specifying such a permission in a Java security policy file is as follows:

```
permission javax.security.auth.PrivateCredentialPermission
➤ "CredentialClassName PrincipalClassName \"principal_name\"",
➤ "read";
```

As an example, we must grant permission to the principal name `edison` for access to their private credentials with our `AuthenticationExample` using the following Java security permission in a Java security policy file:

```
permission javax.security.auth.PrivateCredentialPermission
    "java.lang.String ejava.jaas.MyPrincipal \"edison\"", "read";
```

Alternatively, an asterisk, *, may be used in place of the *CredentialClassName* to indicate that this permission applies to all credential class types. Furthermore, an asterisk may also be used in place of the *principal_name* to indicate that this permission applies to all principals with the associated *PrincipalClassName* as a principal class type. If the *PrincipalClassName* is also specified with an asterisk, the permission applies to any principal class type. As a final note, additional *PrincipalClassName* and *principal_name* pairs can be specified within the same permission.

Java Security Policies with Authentication Permissions

As we have previously mentioned, our example application also requires that we configure additional Java security policies in order to execute our example application. Specifically, the *AuthenticationExample* application creates a *LoginContext* object, attempts to read private credentials for our user named *edison*, attempts to perform a privileged action associated with a *Subject*, attempts to read a properties file from the local file system, and attempts to write a security log file to the local file system. Our Java policy file thus utilized by our example application must have the following additional grant entries:

```
grant codebase "file:example.jar"
{
    permission javax.security.auth.AuthPermission "createLoginContext";
    permission javax.security.auth.PrivateCredentialPermission
        "java.lang.String ejava.jaas.MyPrincipal \"edison\"", "read";
    permission javax.security.auth.AuthPermission "doAs";
    permission java.io.FilePermission "credentials.properties", "read";
    permission java.io.FilePermission "myLogFile.txt", "write";
};
```

As just shown, our authentication example code is packaged into a local *example.jar* file referenced by the codebase in this standard Java security policy file.

Thus, we may then execute our example with the following command whereby all such referenced configuration and policy files are contained in the local directory:

```
java -Djava.security.auth.login.config=jaas.config
➤ -Djava.security.manager
➤ -Djava.security.policy=security.policy
➤ ejava.jaas.AuthenticationExample
```

Because our *security.policy* file also included an entry defined earlier to allow *edison* to write to a log file, the example should execute with no security exceptions. If we were to alter the *security.policy* file to assume a different principal name for *myLogFile.txt* file write permission and execute the application, a *FilePermission* failure for *edison*'s permission to write to the *myLogFile.txt* file would occur.

Using JAAS with EJB

Although the JAAS API previously described can apply to standalone J2SE and container-based J2EE environments alike, the most common usage for JAAS is expected to be in the context of EJB container environments. There are a few reasons to expect this. First, enterprise applications represent the most common application type used with Java. Second, security-critical enterprise applications are often constructed in a demilitarized zone that naturally leverage use of an EJB container/server environment. Hence, a standard means for authentication and authorization in such environments is crucial. This is where JAAS may step in and provide the greatest help.

EJB Clients and JAAS

An EJB client first must obtain information that it will use to authenticate itself with the server. This information may be gotten in various ways, including from a command line, a GUI, or perhaps a local store of credential information.

The EJB client may choose to implement a `CallbackHandler` object as previously described to retrieve login information from a user. A client uses the `LoginContext` abstraction to associate such a `CallbackHandler` with the client-side JAAS authentication process. A custom or vendor-supplied class that implements the `LoginModule` interface must then be set by the `LoginContext` with the `CallbackHandler`. The custom or vendor-supplied `LoginModule` is then responsible for transferring authentication information stored in the `CallbackHandler` to the EJB server transparent to the EJB client. The EJB client simply uses standard JAAS interfaces and specifies the configuration of the custom or vendor-specific `LoginModule` via the JAAS configuration file. The EJB client has no need to talk directly with the `LoginModule` implementation class.

Note

An example EJB client using JAAS is associated with the code for this chapter located as described in Appendix A. The code is extracted to the `examples/src/ejava/ejb/security/jaas` directory. A `TestClient` class in that directory represents a JAAS client that talks to the `OrderManagerEJB` session bean deployed to the EJB server. The `TestClient` uses a `TestCallbackHandler` for authentication. Note that this example assumes that you are running the BEA WebLogic Server and needs to make use of a vendor-specific `URLCallback` class. Such a vendor binding was necessary at the time of this writing due to the early stages of integration of JAAS with J2EE. A `TestAction` associated with these examples performs `PrivilegedAction` operations on the `OrderManagerEJB` that requires security-critical access control decision making.

J2EE application clients may also specify a special `<callback-handler>` element inside of their `application-client.xml` deployment descriptor as defined in Chapter 23, “Enterprise Application Services.” This element is defined with a fully qualified class name of a `CallbackHandler` implementation as illustrated here:

```

<application-client ...>
  ...
  <callback-handler>
    ejava.ejb.security.jaas.TestCallbackHandler
  </callback-handler>
  ...
</application-client>

```

The `<callback-handler>` element references a `CallbackHandler` interface implementation with a parameterless constructor. The J2EE application client's container uses this callback handler to extract authentication from the client at runtime before the client is allowed to interact with a security-critical EJB.

EJB Servers and JAAS

On the server side, the EJB server makes any vendor-specific calls to the server's underlying authentication mechanisms. The EJB server vendor may use JAAS on the server side as well to transform authentication information into concrete `CallbackHandler` objects usable by that vendor's EJB container. JAAS may be used to initialize a `LoginModule` implementation on the server side with the `CallbackHandler` associated with the client's authentication information. The `LoginModule` implementation may be provided by your EJB container vendor or it may be a custom `LoginModule`. If your EJB container vendor allows you to implement your own custom JAAS `LoginModule` classes, you may generally implement them in a vendor-independent fashion by adhering to the JAAS standard interfaces. Thus, you'll be able to use those custom `LoginModule` implementations in other J2EE containers supporting JAAS.

Custom JAAS `LoginModule` Classes

Custom `LoginModule` classes may be needed if your vendor's supplied `LoginModule` implementations cannot work with your particular credential storage mechanism. For example, you may store legacy credential information in a database that uses a proprietary encryption algorithm to decrypt passwords. In such a case, even if your EJB vendor supplies you with some generic `LoginModule` implementation for retrieving information from a database, it may not be able to support the decryption algorithm needed for your custom authentication process.

The EJB server then uses JAAS to induce a login that performs the `LoginModule` class-specific authentication process. If a successful login occurs, the JAAS subject information is associated on the server side with the client's thread or by some other security context association means. One means is to sign the principal information associated with an authenticated `Subject`. Such signed "security tokens" may then be returned to the EJB client as security context information. The server can then check such security tokens in a much more efficient fashion whenever subsequent security-critical EJB operations are attempted by the client.

Conclusions

Thinking about the reliability, availability, scalability, and security of your EJB applications means thinking about EJB assurance. This chapter described some of the most core technologies available to you from within J2EE as well as provided by vendors to help you build assured EJB applications. Use of transactions with EJB can be done in either a programmatic or a declarative form and helps guarantee the reliability of your EJBs in distributed enterprise environments. The scalability and availability of your EJB applications is largely a function of what clustering features your EJB container provides to support fail-over and load balancing. Securing your EJBs involves the most consideration. Standard provisions for EJB security largely revolve around programmatic and declarative access control mechanisms. Although JAAS may be used to help provide standard authentication mechanisms, other vendor-specific authentication and security mechanisms may be employed. Regardless, this chapter leaves you with the basis to think about how to approach infusing assurance into your EJB applications and how you might do this leveraging J2EE standards as much as possible.