CHAPTER 28

Extending ColdFusion with CORBA

IN THIS CHAPTER

For both developers and companies, a common goal has been to reuse business logic for both internal and external applications. One major problem is that each organization standardizes and writes components on different platforms (such as Unix or NT) and in different development languages (such as Java, C++, or Ada). There needs to be a way for all of these component objects to communicate, appearing as a single application running on a single machine regardless of the language, host, or environment. One standard that has emerged to address this need is Common Object Request Broker Architecture (CORBA).

Introduction to CORBA

CORBA, basically a specification for building and using cross-platform distributed software objects, was created by the Object Management Group (OMG) in 1989. This allows a `PassengerManager` component written in Java, for instance, to be accessed by a C++ client, with the C++ client not knowing where the `PassengerManager` component is hosted or in what language it was written. The details on how this is possible will be discussed in this chapter. CORBA's mission, then, is somewhat similar conceptually to the mission of Web Services as discussed elsewhere in this book.

CORBA requires that all application logic be written in the form of objects, where its functionality can be described through an Interface Definition Language (IDL) definition. It is not required that the language the application logic is written in be object-oriented.

➜ IDL is discussed in the "OMG Interface Definition Language" section.

NOTE

CORBA principles were critical in the forming of the Enterprise JavaBeans standard as well as Java RMI.

As CORBA is a specification, it relies on various vendors and research organizations to provide implementations to host the CORBA objects. These are referred to as Object Request Brokers (ORB). The popular commercial ORBs available today are VisiBroker from Borland (`www.borland.com/visibroker`), Orbix from IONA (`www.orbix.com/products/orbix`), and e*ORB from PrismTech (`www.prismtechnologies.com`).

ColdFusion can communicate with CORBA through the `<cfobject>` tag or the `CreateObject()` function. Support for CORBA is currently only available in the Enterprise editions of ColdFusion.

Who Is the OMG?

The Object Management Group is an organization created in April 1989 by 11 companies (such as Sun Microsystems, IBM, and American Airlines) with the purpose of "creating a component-based software marketplace by hastening the introduction of standardized object software." From the original 11 companies, the OMG has since grown to more than 800 members while still promoting its charter of providing a common framework for application development.

The OMG Web site is `www.omg.org` and contains the most up-to-date specification documentation on CORBA as well as information on the other standards that OMG supports.

How CORBA Works

Although this chapter doesn't cover how to write CORBA objects, you will learn some CORBA basics to help you understand the role that ColdFusion plays and how things work behind the scenes. There are several CORBA concepts and services that I have not covered due to the lack of relevance to the way ColdFusion uses CORBA.

The pieces of CORBA can be simplified into the following three parts: client, Object Request Broker, and object service. Basically, the client (ColdFusion in our case) makes a request for an object to the ORB. The ORB takes this request and invokes the method on the object through the object service. Return values from the method are passed back through the ORB to the client. An example might be of a `BankAccount` object that returns a balance based on the username passed in.

Object Request Broker

The ORB represents the cornerstone of the CORBA architecture. The function of the ORB is to provide communication between the client and the object service, which performs the work. The ORB is responsible for finding the requested object, passing parameters along, invoking the necessary object, and returning the results to the client. The ORB shields the location and the object implementation from the client. This shielding allows the client to transparently request and work with objects that are written in multiple languages and physically located throughout the network or even on the same machine. The client in essence views the entire system as one homogeneous environment, although in reality it may be diverse.

When a client requests an object from the ORB, the ORB returns to the client a reference to the object instance, which resides in the object service. This reference is known as the Interoperable Object Reference (IOR). The IOR contains the information necessary for the ORB to relocate the same object for subsequent requests.

NOTE

Another way to view this is as a client/server architecture. Think of the ORB as the communication mechanism between the client and object service.

Clients use the object references to make requests on the object. These object references can be obtained via directory services or by converting stringified IOR (`.ior`) files. Depending on how the object is requested, a new instance may be created in the object service. For example, if you are using an existing IOR (stringified `.ior` file) when requesting an object, a new object will not be created. However, if you use the Naming Service of the ORB to obtain an object, a new object would be created on the server. Using the Naming Service and existing (stringified) IOR files will be covered later in this section.

In any case, the ORB receives a request for an object and returns it. Notice that the ORB behaves the same by routing the request to the object service through the ORB "server" to the object server for both remote and local objects.

NOTE

Elaborating on the client/server parallel of the client and object service, you can think of each object as having a client interface and a server implementation.

Along with being responsible for locating and routing the request from the client to the corresponding component, the ORB is responsible for transferring data from the client to the server and passing the information back from the server to the client. For the data to be passed through the diverse environments, the data has to be converted into a standard format that can be passed through the network. The format, referred to as Over the Wire format, is done through the process called marshalling. The Over the Wire format is binary. The repackaging of the data into the native format is called unmarshalling.

NOTE

The terms marshalling and unmarshalling also are known as serialization and deserialization, respectively.

With the data being converted into a binary format, the ORB can pass the request onto another ORB server on the network if needed. This action is known as inter-ORB communication. The General Inter-ORB Protocol (GIOP) was created to allow the ORBs to communicate. However, because GIOP is a generalized protocol, it is not actually used. Instead, a protocol based on the GIOP, such as Internet Inter-ORB Protocol (IIOP), is used. IIOP is generally assigned to port 9100. Each ORB on the network is generally configured to listen to this port for requests from other systems. We can see at this point how it is possible for CORBA to allow objects to be hosted on remote machines running different operating systems.

NOTE

It is possible to create other protocols based on GIOP to use instead of IIOP. Be aware that stringified IOR (`.ior` files) are dependent on IIOP to work at all times.

Many ORB services can be used to locate and obtain an object reference. An ORB has an Interface Repository (IFR), which is in essence a database of all the objects, their interfaces, and their locations that the ORB knows about. If the ORB doesn't have knowledge of the object, then the client won't be able to gain access to it, even if it does exist somewhere on the network. The Naming Service is the only ORB service ColdFusion currently interfaces with. The Naming Service allows you to request an object reference through a literal name, such as `BankComponent` or `Macromedia.Transactional.BankComponent`. This is similar to how COM components are generally accessed. After an object's location is determined, an object reference is created.

Another way to obtain an object reference is through its stringified IOR file. This file is basically an object reference encased in a file; the file consists of a string of hexadecimal numbers. An object reference just stores the location of the object in the network, not its interface definition. To work with the object, the object's interface needs to be stored in the ORB's IFR. Because the object's location is encased inside the `.ior` file, the connection performance is slightly better using `.ior` files than using the Naming Service. There is no difference between the two when invoking a method.

NOTE

Because the stringified IOR points to a previously created object instance, the success of the client connection through this `.ior` file depends on that object instance remaining.

Regardless of whether we used an `.ior` file or the Naming Service to obtain an object reference, the ORB uses its DII interface to dynamically build the method invocation on an object.

OMG Interface Definition Language

With the ORB being the cornerstone of the CORBA architecture, the Interface Definition Language is the mortar. The IDL is used for two main things: defining an object's interface and language mappings. The IDL is critical because it neutralizes the language-specific nature of the components and clients by providing a common way an object is viewed as well as the way data is passed.

An object's interface specifies the methods and properties that an object supports. An interface stores what is publicly available for the object, and is similar to `.h` files in C++ and interfaces in Java. The interface of an object is stored in the Interface Repository of the ORB and in an `.idl` file. From there, the clients can gain references and invoke methods.

In the "Object Request Broker" section earlier in the chapter we talked about how a request is made up of data being passed from the client and to the object on the remote server. The data was passed in On the Wire format (binary string). The ORB uses the IDL mapping to do its marshalling/unmarshalling of the data being used in the request.

OMG IDL is a declarative language, not a programming language; as such, it does not provide features such as control constructs, nor is it directly used to implement distributed applications. Instead, language mappings determine how OMG IDL features are mapped to the facilities of a given programming language. Language mappings force interfaces to be defined separately from object implementations. This allows objects to be constructed using different programming languages and yet still communicate with one another. Language-independent interfaces are important within large networks because not all programming languages are supported or available on all platforms. Several language mappings are currently available, such as ones for C, C++, Java, Smalltalk, and Ada. To see the complete list of the language mappings currently available, check the OMG and CORBA Web sites (`www.omg.org` and `www.corba.org`).

In Listing 28.1, we see a simplified IDL file for a conceptual `PassengerBoardingManager` object, living on an unknown host and developed using an unknown programming language. It defines a `Passenger`, a `Flight`, and a `Passport` structure, a `boardingAgentId` property, an enumeration named `seatType`, a type definition called `SeatPreferences` which is composed of a sequence of `seatType` items, and a single interface called `PassengerManager` which contains three methods and an exception called `PMException`. You'll notice that each of the methods raises a `PMException`. Since CORBA is by nature distributed, these exceptions become the means of capturing error information from our objects.

Listing 28.1 The Passenger Boarding Manager IDL File

```
module PassengerBoardingManager {
  struct Passenger {
    string name;
    double ticketNumber;
    Passport passport;
    Flight flight;
  };
  struct Flight {
    string flightId;
    string airline;
    string origin;
    string destination;
  };
  struct Passport {
    string issuingCountry;
    string passportId;
  };
  string boardingAgentId;
  enum seatType {Window, Aisle, None};
  typedef sequence<seatType> SeatPreferences;
  interface PassengerManager {
    exception PMException {
      string msg;
    };
    boolean validTicket( psngr Passenger ) raises (PMException);
    boolean validBoardingPass( psngr Passenger, SeatPreferences ) raises
            (PMException);
    boolean noFlyClear( psngr Passenger) raises (PMException);
  };
              };
```

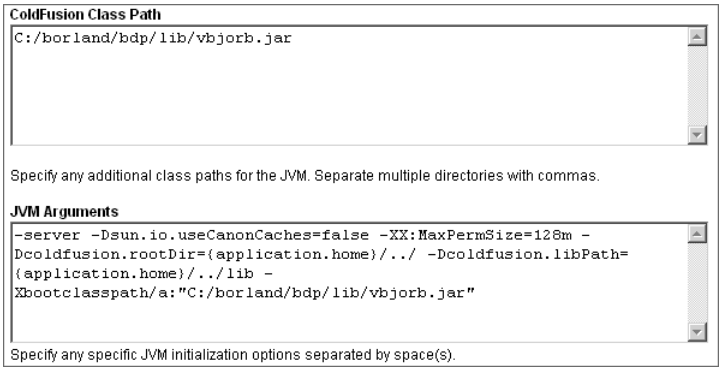
Configuring ColdFusion to Work with CORBA

Prior to ColdFusion 5, ColdFusion had bundled the C++ ORB runtime libraries from Borland (VisiBroker 3.3 on NT/HP-UX and VisiBroker 4.0 on Solaris). These were statically linked to the ColdFusion server executable and, therefore, only allowed you to work with this single ORB and version. ColdFusion 5 introduced a new architecture to allow for dynamic loading of libraries to use as a connector for a specific ORB. This approach is more generic and does not tie the ColdFusion user to a specific ORB vendor or version.

Before you can work with CORBA objects in your ColdFusion pages, you'll first need to add the runtime libraries for your particular ORB to the ColdFusion classpath. The way in which this is accomplished will depend on whether you are using the J2EE configuration or not. If you are, you'll add the `jar` file to your application server classpath, using the method specific to the particular server.

In any case, in the ColdFusion Administrator navigate to Server Settings > Java and JVM. Assuming you are using a Borland VisiBroker product as your ORB infrastructure, you'll first enter the full path to your `vbjorb.jar` file in the ColdFusion Class Path text box. Second, if your JVM is 1.4 or newer, add `-Xbootclasspath/a:"C:/borland/bdp/lib/vbjorb.jar"` to the JVM Arguments text box, assuming the full path to your jar file `C:\Borland\bdp\lib\vbjorb.jar` (Figure 28.1).

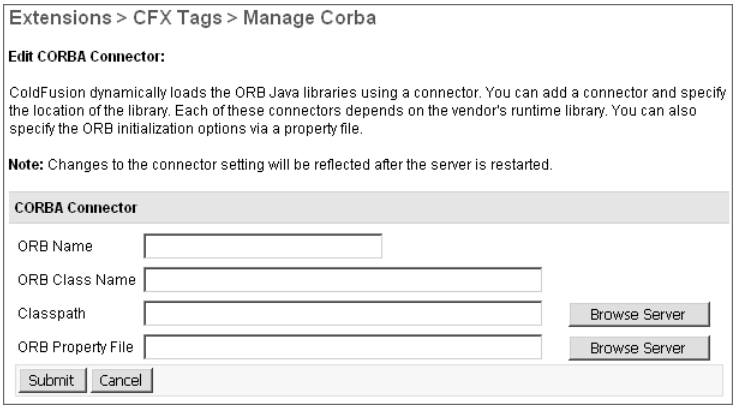
Figure 28.1
The Java and JVM administration screen.



Now that the `vbjorb.jar` file is in the classpath, you need to set up the Extensions > CORBA Connectors page in the ColdFusion Administrator (Figure 28.2).

Register a new CORBA connector with an ORB Name of `visibroker`, an ORB Class Name of `coldfusion.runtime.corba.VisibrokerConnector`, and an ORB Property File that points to the location of the `CFusion/lib/vbjorb.properties` file (the full location would be `c:\CfusionMX7\lib\vbjorb.properties` in a typical Windows installation of ColdFusion MX7). Leave the Classpath textbox empty, and submit your connector information. From the CORBA Connectors page select your newly registered connector. Finally, restart your ColdFusion server.

Figure 28.2
The CORBA administration screen.



Working with CORBA in ColdFusion

The `<cfobject>` tag and the `CreateObject()` function are used by ColdFusion as the gateway to the CORBA world. These two mechanisms for working with CORBA are identical, and both will be used in the remaining examples. As discussed in earlier sections, the client needs to make a request to the ORB for an object reference before it can invoke an object method. The job of the `<cfobject>` tag and `CreateObject()` function is just that—get an object reference. The object reference stored in the ColdFusion variable returned from the `<cfobject>` tag is then used to invoke methods against the remote object.

NOTE

It is possible to store the object reference inside a persistent variable such as `application.PassengerManager`. It is important to note that this doesn't store the data about the object, just the object's IOR. Each request still needs to go through the ORB to be executed.

The following is the syntax of the `<cfobject>` tag and `CreateObject()` set to use CORBA. See Table 28.1 for a listing of all the attributes used by `<cfobject>` and `CreateObject()`:

```
<cfobject
 action="Create"
 type="CORBA"
 context="context"
 class="file or naming service"
 name="myObject"
 locale="">
```

and:

```
myObject = CreateObject( type, class, context, locale)
```

Table 28.1 Attributes for `<cfobject>` and `CreateObject()`

NAME	DESCRIPTION
`type`	Required. Specifies the object type to be accessed. May be `COM`, `Java`, or `CORBA`. This must be set to CORBA for ColdFusion to connect to the CORBA server. See Chapters 26 and 29 for `COM` and Java usage, respectively.
`context`	Required. Must be set to `IOR` or `NameService`. `IOR` causes ColdFusion to use the Interoperable Object Reference (IOR) stored in an `.ior` file to access the object reference. `NameService` causes ColdFusion to use the Naming Service of the ORB to get an object reference.
`class`	Required. Specifies different information, depending on the `context` specification. If `context` is `IOR`, this attribute specifies the name of a file that contains the stringified version of the `IOR`. ColdFusion must be able to read this file at all times; it should be local to the ColdFusion server or on the network in an open, accessible location. Since ColdFusion MX, if `context` is `NameService`, this attribute specifies a forward slash-delimited (/) naming context for the Naming Service, such as `Macromedia/Transactional/Account`. Prior to ColdFusion MX, this was a period-delimited naming context.
`name`	Required. Specifies the name for the variable that will be created to hold the reference to the CORBA object. Your application uses this to reference the CORBA object's methods and attributes.
`locale`	Optional. If specified, this attribute must be the name of an ORB registered in the ORB Name field of the CORBA Connector screen in the ColdFusion MX7 Administrator.

In order for us to discuss an example of how this fits together, we'll need to know about one additional IDL module containing the `FlightManifestManager` module, which like the `PassengerBoardingManager` in Listing 28.1, defines a `Passenger`, a `Flight`, and a `Passport` structure, and a single interface. There are no `seatTypes` or `SeatPreferences` items defined in this module, though. The interface looks like this:

```
interface FlightManifestManager {
  exception FMException {
    string msg;
  };
  boolean addPassengerToFlightManifest( psngr Passenger, fl Flight ) raises
(FMException);
};
```

Now we are ready to see how all of this pulls together in Listing 28.2, our conceptual airline passenger boarding application.

Listing 28.2 Working with CORBA Objects in Our ColdFusion Page

```
<!---
 Filename: PassengerBoardingManagerApplication.cfm
 Purpose: Demonstrates use of CORBA objects in ColdFusion pages
--->
<!---
```

Listing 28.2 (continued)

```
        Enable application variables. We would normally do this in the
        application.cfc file. Prior to ColdFusion MX 7, we normally did
        this in the application.cfm file.
 --->
<cfapplication name="MyCorbaApp">

      <!-- static HTML used to set the display -->
      <html>
        <head>
          <title>CORBA Example</title>
        </head>
        <body>

        <table>
          <tr>
            <td colspan="2">Passenger Boarding Manager Application</td>
          </tr>

  <cftry>
    <!-- Check to see if there is already a reference -->
    <!--   to the remote object stored -->
    <cflock scope="APPLICATION"
            throwontimeout="Yes"
            timeout="10"
            type="ReadOnly">
      <cfif not IsDefined("APPLICATION.PassengerManager")>
        <cfset NewPMNeeded="1">
      <cfelse>
        <cfset NewPMNeeded="0">
        <cfset PassengerManager=APPLICATION.PassengerManager>
      </cfif>
    </cflock>

    <!-- If there is not a reference already stored then create -->
    <!--   a new reference and store it for re-use -->
    <cfif NewPMNeeded>
      <cflock scope="APPLICATION"
              throwontimeout="Yes"
              timeout="10" 5w
              type="EXCLUSIVE">
        <cfobject action="Create"
                  type="CORBA"
                  name="APPLICATION.PassengerManager"
                  class="C:\corba\ior\PassengerManager.ior"
                  context="IOR"
                  locale="">
        <cfset PassengerManager=APPLICATION.PassengerManager>
      </cflock>
    </cfif>

    <!--- the "Flight" struct ---->
    <cfset fl=StructNew()>
    <cfif IsStruct(fl)>
      <cfset fl.flightId="cf2318">
      <cfset fl.airline="ColdFusionAirways">
```

Listing 28.2 (CONTINUED)

```
  <cfset fl.origin="tmp">
  <cfset fl.destination="ord">
</cfif>

<!--- the "Passport" struct ---->
<cfset pass=StructNew()>
<cfif IsStruct(pass)>
  <cfset pass.issuingCountry="US">
  <cfset pass.passportId="042534534654769878235622987456">
</cfif>

<!--- the "Passenger" struct ---->
<cfset psngr=StructNew()>
<cfif IsStruct(psngr)>
  <cfset psngr.name="Adam Smith">
  <cfset psngr.ticketNumber="8708963543">
  <cfset psngr.passport=pass>
  <cfset psngr.flight=fl>
</cfif>

<!-- the SeatPreferences sequence needed by validateBoardingPass -->
<cfset seatprefs=ArrayNew(1)>
<cfset seatprefs[1]="1">
<cfset seatprefs[2]="0">
<cfset seatprefs[3]="2">

<!-- output some basic passenger information -->
<cfoutput>
      <tr>
        <td>Name:</td>
        <td>#psngr.name#</td>
      </tr>
      <tr>
        <td>Flight:</td>
        <td>#fl.flightId#</td>
      </tr>
      <tr>
        <td>Destination:</td>
        <td>#fl.destination#</td>
      </tr>
</cfoutput>

<!--perform a series of validations on the passenger -->
<!-- using the PassengerManager object via CORBA -->
<cftry>
  <cfif PassengerManager.validTicket(psngr)>
    <cfif PassengerManager.validBoardingPass(psngr)>
      <cfif PassengerManager.noFlyClear(psngr)>
        <cfset passengerValidated="1")>
      </cfif>
    </cfif>
  </cfif>
  <!-- be sure to catch any exception thrown by the remote object -->
  <cfcatch TYPE="coldfusion.runtime.corba.CorbaUserException">
    <cfoutput>
```

Listing 28.2 (continued)

```
            <tr>
              <td COLSPAN="2">
                Unable to verify information for #psngr.name#
                onboard flight #fl.flightId#. <br/>
                The passenger may <em>not</em> board the airplane!
              </td>
            </tr>
          </cfoutput>
          <cfset exceptionStructure=cfcatch.getContents()>
          <cftrace var="exceptionStructure"
                   inline="No"
                   type="Error"
                   category="CORBA"
                   abort="No">
        </cfcatch>
      </cftry>

      <!-- if the passenger validates, -->
      <!--   add to the manifest       -->
      <cfif passengerValidated>
        <!-- get an object reference to the FlightManifestMananger -->
        <cftry>
          <cfset FlightManifestManager=CreateObject("CORBA",
                   "TSA/International/Manifest/FlightManifestManager",
                   "NameService")>
          <!-- be sure to catch any exception thrown by
          <!--   the remote object -->
          <cfcatch type="Object">
            <cfoutput>
              <tr>
                <td COLSPAN="2">
                  Unable to add #psngr.name# to the flight #fl.flightId#
                  manifest. The system is unavailable at this time.<br/>
                  The passenger may <EM>not</EM> board the airplane!
                </td>
              </tr>
            </cfoutput>
            <cfset exceptionStructure=cfcatch.getContents()>
            <cftrace VAR="exceptionStructure"
                  inline="No"
                    type="Error"
                 category="CORBA"
                    abort="No">
          </cfcatch>
        </cftry>
        <!-- get an object reference to the FlightManifestMananger -->
        <cftry>
          <cfset addedToManifest=
            FlightManifestManager.addPassengerToFlightManifest(psngr,fl)>
          <cfif addedToManifest>
            <cfoutput>
              <tr>
                <td COLSPAN="2">
                  Passenger may board now.
                </td>
```

Listing 28.2 (CONTINUED)

```
            </tr>
          </cfoutput>
        <cfelse>
          <cfoutput>
            <tr>
              <td COLSPAN="2">
                Passenger may <em>not</em> not board the airplane!
              </td>
            </tr>
          </cfoutput>
        </cfif>
        <!-- be sure to catch any exception thrown by ‡
        <!--   the remote object -->
        <cfcatch TYPE="coldfusion.runtime.corba.CorbaUserException">
          <cfoutput>
            <tr>
              <td COLSPAN="2">
                Unable to add #psngr.name# to the flight #fl.flightId#
                manifest.<br/>
                The passenger may <em>not</em> board the airplane!
              </td>
            </tr>
          </cfoutput>
          <cfset exceptionStructure=cfcatch.getContents()>
          <cftrace var="exceptionStructure"
                   inline="No"
                   type="Error"
                   category="CORBA"
                   abort="No">
        </cfcatch>
      </cftry>

    <!-- else if the passenger did not validate -->
    <cfelse>
      <cfoutput>
        <tr>
          <td COLSPAN="2">
            #psngr.name# may <em>not</em> board flight #fl.flightId#
            to #fl.destination#.<br/>
          </td>
        </tr>
      </cfoutput>
    </cfif>
    <!-- catch any exception we experienced getting the -->
    <!--   PassengerManager CORBA object reference      -->
    <cfcatch type="Object">
      <cfoutput>
        <tr>
          <td COLSPAN="2">
            The Passenger Boarding Manager is experiencing connectivity
            issues and is unavailable at this time.
          </td>
        </tr>
      </cfoutput>
      <cfset exceptionStructure="cfcatch.Detail">
      <cftrace var="exceptionStructure"
```

Listing 28.2 (continued)

```
                inline="No"
                type="Error"
                category="CORBA"
                abort="No">
      </cfcatch>
    </cftry>
          <!-- close up the HTML tags -->
          </table>
        </body>
      </html>
```

The first thing that we do is drop in some generic HTML to handle creating and opening a table which will hold the results of the code processing. Next, we surround the entire CORBA interaction with a `<cftry>` block. We will cover exceptions in a moment. Following that we check to see whether a `PassengerManager` object reference is stored in the application scope. If it isn't, we grab a reference to the `PassengerManager` using an `.ior` file. By using the `APPLICATION` scope, we can persist and share the object connection throughout our many requests thereby eliminating the overhead of repeatedly establishing a connection to the same object. We could have alternatively used the session and server scopes to accomplish this:

```
<cfobject action="Create"
          type="CORBA"
          name="APPLICATION.PassengerManager"
          class="C:\corba\ior\PassengerManager.ior"
          context="IOR"
          locale="">
```

NOTE

ColdFusion doesn't support the IDL `module` type. Instead we dereference CORBA IDL interfaces as objects.

Next, using the `StructNew()` function, we create the structures we'll need to interact with the CORBA object methods before. Those are: `Flight`, `Passport`, and `Passenger`. Additionally, we create the `SeatPreferences` sequence, `seatprefs`. In Listing 28.1, the IDL for the `PassengerBoardingManager`, `seatType` is defined as an enumeration. ColdFusion considers an enumeration to be a zero-indexed integer. Hence, the Passenger's ordered seating preferences are: Aisle, Window, and None.

We use `<cfoutput>` to display some information from our structures inside a table, which will eventually display a result letting the boarding agent know whether or not to allow the passenger onboard the airplane. The code executes each of the three methods by the `PassengerManager` interface, passing in our `Passenger` structure. In this case, we do that by using a series of nested `<cfif>` tags because all of the methods return a Boolean value, and doing so makes clear some of the implied logic of our example application. If each of the methods returns `true`, the `passengerValidated` variable is set:

```
<cfif PassengerManager.validTicket( psngr ) >
  <cfif PassengerManager.validBoardingPass( psngr ) >
    <cfif PassengerManager.noFlyClear( psngr ) >
      <cfset passengerValidated = 1) >
    </cfif>
  </cfif>
</cfif>
```

In each of these method calls, we've passed the input parameter by value, not reference. CORBA `in` parameters are passed by value. Had these parameters been `out` or `inout` parameters of the method, CORBA would have expected a reference to the value. If you pass a variable by reference and the CORBA object modifies its value, the value also changes on your page.

We now check to see if the passenger's information validated. Assuming it did, we create a reference to another CORBA object using the `CreateObject()` function:

```
<cfset FlightManifestManager = CreateObject("CORBA",
               "TSA/International/Manifest/FlightManifestManager",
               "NameService") >
```

Since we encountered no problems creating a reference to the remote object, we make a call to its `addPassengerToFlightManifest()` method, passing in our `Passenger` and `Flight` structures. Based on the Boolean return value, we output a friendly message letting the boarding agent know whether the passenger can proceed (Figure 28.3).

Figure 28.3

The results of Listing 28.2.

Passenger Boarding Manager Application	
Name:	Adam Smith
Flight:	cf2318
Destination:	ord
Passenger may board now.	

In addition to accessing the methods of an object, ColdFusion supports accessing a CORBA object's public properties. This is also done through dot notation. So the `boardingAgentId` property of the `PassengerManager` reference would be accessed as `PassengerManager.boardingAgentId`. Because I frequently use camel case (capitalizing the first letter of each word but placing no spaces between words) for many of my variables and method names, it is important to note that though ColdFusion is case-insensitive, CORBA is not. So when designing your CORBA objects, it is best to avoid method names which differ only in case. If an interface defines both `getMyResult()` and `getmyresult()`, you can never be certain which method ColdFusion will call!

Handling CORBA Exceptions

In Listing 28.2 we wrapped the entire code that deals with the CORBA interaction in a `<cftry>` block. Additionally, each time we accessed a CORBA object or created a new one, those too were wrapped in `<cftry>` blocks. Though it is always good development practice to include robust error handling, it is especially poignant here because CORBA relies on external resources which may or may not be available. By using error handling, we can capture and gracefully handle errors that occur while connecting to and using CORBA objects.

When an error occurs while connecting to or using a CORBA object, ColdFusion throws an error of type `Object`.

```
<cferror type="Exception"

        exception="object"
         template="../common/objexcept.cfm"
           mailto="admin@thiscompany.com">
```

When using the <cftry> block, additional error information may be provided by the CORBA mechanism; if so, it is available in a <cfcatch> block via the cfcatch.getContents() function as shown in Listing 28.2. The getContents() function returns a structure that contains formatted information about the exception as reported by the IDL.

➜ For details, please refer to the ColdFusion MX 7 documentation.

You will notice that the <cfcatch> tags following operations where we created references to CORBA objects have a type attribute of Object. This is because ColdFusion raises an exception of type Object when there is an error creating a reference to a CORBA object. To ensure the capture of any exceptions raised by the remote object, we set the type attribute of other <cfcatch> tags to coldfusion.runtime.corba.CorbaUserException. Since it isn't desirable for the user to see a stack trace or exception information spewed onto the screen, we use the <cfcatch> block to output a friendly message for the user, and then dutifully write the contents of the exception to the log file.

In Listing 28.2, we extracted a meaningful set of data about the exception using <cftrace>. However, we could have also used <cfdump var="#exceptionStructure#">. Since the latter will write to the log each time it is called, it isn't a good option for a production application. Regardless of how we choose to utilize the error information, robust error handling is a requirement in distributed computing environments where network outages and bottlenecks can otherwise cause your ColdFusion applications to appear ill-behaved. A few well-placed <cftry> blocks, provide the basis for both simple debugging during development and graceful handling of undesirable runtime results.