Copyright © 2002, 2003 OpenEAI Software Foundation.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Section being the first section entitled "Introduction", with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled "Appendix 1: GNU Free Documentation License".
Introduction

Many times, developing an application is not the most difficult task associated with the entire application development cycle. Once an application is developed and ready to run, it must be deployed. This can prove to be quite challenging in complex computing environments. Here, we describe recommended patterns for deploying applications based on the OpenEAI Message Protocol and the supporting OpenEAI APIs. That said, we recognize that most of these patterns are not specific to OpenEAI. They are patterns that can be used for many types of application deployment. Many of these patterns are simply an attempt to address requirements shared by most deployments regardless of their specific technology or purpose. The only thing that makes them specific to OpenEAI is that we’ll be discussing specific artifacts used by OpenEAI-based applications and gateways.

From the beginning, we have designed OpenEAI to reach several major deployment goals. These include consistency, ease of upgrade and maintenance, scalability and performance, high availability, and durability. Many applications developed with the OpenEAI APIs are intended to be long-running, for example, messaging gateways, scheduled applications and servlets. The objects these applications require at runtime should be configured and initialized when the application starts, and available to the application for its entire life. The OpenEAI foundation APIs have been developed with this as one of its guiding parameters. However, an underlying API can only work as intended if the techniques used to deploy the application are themselves sound. There are many things beyond the control of any single developer. While the OpenEAI APIs have been designed to be as flexible as possible, there are many choices that can be made that will make life easier on those who have to support OpenEAI applications down the road. This document will discuss some of these requirements as well as techniques that have been used to address them to date.

We’ll focus primarily on the artifacts that make up an OpenEAI application, or a gateway. Through discussion of these artifacts and the patterns used to deploy them, it will hopefully become apparent why these choices are important when deploying an OpenEAI messaging enterprise.
Throughout this discussion, we'll make specific reference to naming conventions, directory structures, required users, etc. Most of these artifacts have several things in common. These artifacts are intended to be accessible from many different locations (location independent), they are intended to be stored in a version control system, and they are intended to be deployed “by environment”. This means, OpenEAI recommends that you set up multiple environments in which to run your applications. This typically includes at least one development environment, one test environment and the production environment. The number of environments is only limited by the requirements driving the need for them. There can be as many environments as necessary to address an organization’s needs.

The target platforms we will discuss include Sun Solaris, Linux, and Windows.

Please note that experienced application deployment staff may prefer to deploy their applications, even OpenEAI applications, in a completely different manner from that described here. There are certainly many different ways to accomplish the same goals successfully. However, one of the goals of the OpenEAI Project is to provide clear, original thinking, including requisite documentation, for the theoretical and practical aspects of implementing integrations. Deploying applications and components in support of enterprise integration is a significant goal of the OpenEAI Project. Outlining at least one strategy for managing the deployment of messaging applications and gateways, establishing jargon, and setting a common frame of reference for discussions about deployments is an important step in initiating our future discourse.

Deployment Pattern Overview

The following is an overview of the recommended pattern for deploying the artifacts that will be discussed throughout this document. Each of these items will be discussed in detail below. The Getting Started with OpenEAI Document takes you through using the applications and gateways that make up this “Example” messaging enterprise. The figure below is a local checkout of a structure that exists in the versioning control system used by the OpenEAI Project. This example describes just one environment. Other environments will have this same, or very similar, structure associated with them. The artifacts associated with an environment or a directory structure can evolve over time. This section only addresses the items used to start and configure OpenEAI-based applications and gateways.
Here is an example of a directory tree which shows application and gateway configuration artifacts:

![Directory Tree Diagram]

The following illustration is an overview of the message tree structure which includes all analysis artifacts developed when integrations are designed. These artifacts are stored in a versioning system and used by the foundation components API as “primed” documents for messages and are used to validate enterprise messages created by application.
Analysis by-products (messaging artifacts):

Messages supported by "any-erp-vendor.com" DTDs and Sample XML

Messages supported by "any-openeai-enterprise.org" DTDs and Sample XML
Artifacts

JAR files

The OpenEAI Java APIs, like many other Java APIs, depend on other freely-available runtime libraries. All of these libraries are included with the OpenEAI distribution. This table lists the library versions used as of now. More information about these APIs is available at the URLs listed. In the future, the OpenEAI APIs will be updated by the OpenEAI Project’s current members and by its new contributors, who will exploit the features of new versions of these supporting APIs.

<table>
<thead>
<tr>
<th>Library</th>
<th>Version</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activation.jar</td>
<td>1.0.2</td>
<td><a href="http://java.sun.com/beans/glasgow/jaf.html">http://java.sun.com/beans/glasgow/jaf.html</a></td>
</tr>
<tr>
<td>Collections.jar</td>
<td>JDOM 1.0 beta 7</td>
<td><a href="http://www.jdom.org/dist/binary/">http://www.jdom.org/dist/binary/</a></td>
</tr>
<tr>
<td>Gnu-regexp-1.0.8.jar</td>
<td>1.0.8</td>
<td><a href="http://www.gnu.org">http://www.gnu.org</a></td>
</tr>
<tr>
<td>Jdom.jar</td>
<td>JDOM 1.0 beta 7</td>
<td><a href="http://www.jdom.org/dist/binary/">http://www.jdom.org/dist/binary/</a></td>
</tr>
<tr>
<td>Log4j.jar</td>
<td>1.0.4</td>
<td><a href="http://jakarta.apache.org/log4j/docs/earlier.html">http://jakarta.apache.org/log4j/docs/earlier.html</a></td>
</tr>
<tr>
<td>Openeai.jar</td>
<td></td>
<td><a href="http://www.openeai.org">http://www.openeai.org</a></td>
</tr>
<tr>
<td>Xerces.jar</td>
<td>1.4.3</td>
<td><a href="http://xml.apache.org/dist/xerces-j/old_xerces1/">http://xml.apache.org/dist/xerces-j/old_xerces1/</a></td>
</tr>
</tbody>
</table>

Note: There are a couple important libraries missing from this list. The first is the JMS Provider. OpenEAI applications can be developed and compiled without any specific JMS Provider libraries because the JMS APIs (Producers and Consumers) within OpenEAI are written entirely to the JMS specification. They don’t leverage any proprietary JMS provider features. This is good. However, to actually run an OpenEAI-based application, there does need to be a set of JMS Provider libraries included in the Java classpath at runtime. These libraries actually implement the JMS specification on behalf of the provider. Additionally, there needs to be administered objects and supporting JMS provider instances running and configured according to the applications being developed. See the Getting Started with OpenEAI Document for examples of an entirely open enterprise including a JMS Provider.

Another library that is missing is the organization's Message Object API (MOA). Typically, there is one JAR file that contains all Java Message Objects supported by an organization. This means all message objects the organization supports and all releases of these objects (1.0, 1.1, 2.0 etc.). If you have two or more different organizations using OpenEAI to integrate a number of systems, you would have two or more JAR files corresponding to each organization's MOA. These JAR files include the Java message objects that were generated based on the analysis
done by each organization. An organization’s MOA is, in effect, the API that an organization exposes to itself or to others that allows developers to build applications to message with their systems.

Examples of these Java message objects include things like a `BasicPerson` Java object, an `EmergencyContact` Java object, etc. Depending on the application being developed, these JAR files could be needed both at runtime and when the application is being developed as well. These JAR files, as well as any JAR files containing the actual applications and gateways, must be included in the classpath at runtime. Refer to the OpenEAI API Introduction Document for details on Java message objects, gateways, and applications.

If your organization has high-availability requirements, you will most likely want to deploy messaging applications and gateways on multiple servers, for two primary reasons. First, in the case of hardware failure, you will want the services provided by your messaging gateways and applications to continue to be available. With multiple instances of messaging gateways and applications deployed on two or more servers, you reduce the possibility of service outages due to hardware failure. Second, all organizations must perform hardware and operating system upgrades and maintenance. With multiple instances of messaging gateways and applications deployed on two or more servers, you allow your server support staff to take one instance of the hardware down without interrupting the services these gateways and applications provide. This practice also allows server support staff to perform these maintenance tasks during business hours, not during off-peak times like nights and weekends.

If your organization has, or if you anticipate it will have, high-performance requirements for a high volume of messaging transactions, there is an additional reason to deploy messaging gateways and applications on multiple servers. Messaging gateways and applications can be effectively scaled up by adding more independent server units by adding additional messaging gateways and messaging application instances to the deployment. This allows organizations to start out with a few, low-cost server units to achieve the high availability goals outlined above and service the initial volume of messaging transactions. Then, as demand for more messaging applications emerges or as messaging volume for existing applications rises, your organization can service higher volume with additional low-cost server units with more instances of messaging gateways and applications.

Given that you will most likely be deploying messaging gateways and applications on multiple servers, you’ll need a way to deploy these artifacts to all of these servers. Ideally, you would like to have one place at which you can change deployment artifacts, and have your changes propagated automatically to all target servers. Not only will these applications exist on servers that are considered “Production”, they may be deployed on servers that are used for development, testing, demonstration, research, etc. Additionally, many developers like to develop things locally, in their own environment first, before deploying the application itself to an “environment”.

As with everything else, these JAR files should be placed in a version control system.

Sample Deployment structure:

```
Environments/Development/Jars/1.0/openeai.jar
Environments/Development/Jars/1.0/any-erp-vendor-moa.jar
Environments/Development/Jars/1.0/any-openeai-enterprise-moa.jar
Environments/Development/Jars/1.0/any-openeai-enterprise-gateways.jar
Environments/Development/Jars/1.0/jndi.jar
```
Note the use of the environment level. Also note that there is a version number associated with each release of these JAR files. This is so that you can specify which version or "constellation" of the JAR files you wish to use when you start an application or gateway. Typically, the things that will change from version to version are the actual applications or gateways being developed. But it may also include things such as new versions of the JNDI specification, for example, or even a new version of OpenEAI. This naming system allows you to have entirely isolated constellations of all JAR files deployed and in use by different applications in the same environment, if necessary. At runtime, you can specify which version you want to use. This practice helps isolate you from change. You may deploy some applications with one constellation of JAR files, that may include a 3.0 version of the OpenEAI APIs, version 2.0 of your preferred JDBC driver, etc., and work with it. A month or two down the road, you may want to develop and deploy some additional applications, and you see there is a new version of OpenEAI, say 4.0, and also a new version of the JDBC driver from your DBMS vendor. You may wish to use these newer libraries for your new applications, but you don’t necessarily want to take the time to change your existing applications to use these new libraries and test them all again, because they are working just fine. So this practice of versioning a constellation of JAR files and placing them in the deployment tree with this version number or JAR set identifier helps isolate you from change.
Application Deployment Descriptors

OpenEAI deployment descriptors (XML documents) are used to configure OpenEAI applications and gateways. These documents contain all the information an application or gateway needs to configure itself. Refer to the OpenEAI API Introduction Document for more information regarding the deployment descriptors.

These documents can be stored in several different places. Typically (especially when an application is being deployed in a production environment), these should be stored in a secure location, because they may contain security credentials needed to connect to databases, directory services, or other data repositories within your organization. The OpenEAI Project recommends that these documents be stored in a secure directory server. Then, they can be retrieved securely via SSL when the application starts, inform the application about the objects it will need to configure, instantiate, and start. During a development cycle, the documents may be stored on a local file system or on a web server but once the objects are being used by the application or gateway to connect to production data repositories security is of the utmost importance.

While many deployment descriptors may be rolled up into one single document for the purpose of documenting an entire messaging enterprise in one place, it is generally recommended that these documents be broken out into individual files when they are actually deployed in a production environment, because different applications contain different security credentials to secured resources. If all these security credentials were contained within a single document, it would mean that any one with access to the one document containing the credentials could gain access to all security credentials for the entire enterprise.

This is likely not desirable. It’s generally better to grant access to a deployment descriptor on an application-by-application basis, meaning that each application uses its own directory server credentials to access only the deployment descriptor it needs.

Regardless of where the deployment descriptors are stored when they are used, they should also be stored in a version control system so that changes can be tracked and backed out if necessary. OpenEAI recommends a structure for this practice.

For example:

```
Environments/Development/Deployments/AevGateway.xml
Environments/Development/Deployments/SelfServiceApplication.xml
Environments/Development/Deployments/SelfServiceServlet.xml
Environments/Development/Deployments/EnterpriseApplicationService.xml

Environments/Test/Deployments/AevGateway.xml
Environments/Test/Deployments/SelfServiceApplication.xml
Environments/Test/Deployments/SelfServiceServlet.xml
Environments/Test/Deployments/EnterpriseApplicationService.xml
```

Note the use of the Environments level to distinguish the configuration artifacts of each environment. Additionally, if you maintain this structure in a version control system, you are able to establish which version of the deployment descriptor was in use in each environment at any given time.
As with JMS Administered objects (listed below), OpenEAI recommends that deployment descriptors be stored in a directory server and accessed via JNDI/LDAP. Since these documents can contain secret credentials needed to access secure devices, we recommend that they not be accessed "in the clear", but via SSL or another secure protocol. The OpenEAI foundation APIs that read these documents are equipped to connect via SSL if the appropriate libraries are supplied and the appropriate certificates have been created.

The following is an example directory tree that illustrates storing these artifacts in a directory server:

![Directory tree diagram](image)

Using a tools such as LDAPBrowser (shown above), the entire deployment descriptor can be copied into a "configxml" object in the directory server. The configxml object class is an object class created for use in posting deployment descriptors. In "directory-schema-speak" it inherits from the object class "top" and has one required attribute of type DirectoryString called "configXmlName" and one allowable attribute of type DirectoryString called "configXmlDocument". It must be added to the directory server schema to use it. These are the locations specified in the "providerUrl" property associated with an application. See the OpenEAI API Introduction Document for more information about application property files.

**Enterprise object documents (EOs)**

Enterprise Object (EO) documents (XML files) are used to specify enterprise business rules on each field within an enterprise Java object. They are used by the OpenEAI Message Object API foundation to apply those rules as data is being set via the various member data methods on those objects. These rules include: translations, data scrubbing, length, data type, formatting, and verification of required fields. See the OpenEAI API Introduction Document for more information regarding the MOA and EO documents, and their use.
The EO documents are initially generated when an organization generates its Message Object API (MOA). These are the business objects that the organization supports throughout the integrations that it has implemented. The rules can be extended during integration analysis.

Like most other artifacts used by OpenEAI applications, these can be stored in a number of locations. During a development cycle, it is common to store these files locally while the application is being developed and while business rule analysis and testing is still occurring. Once these rules have been solidified, it is recommended that they be stored on a web server and accessed from there. This way, you don’t have to deploy the documents to numerous locations. You can simply refer to the web site. These documents do not contain sensitive information, they simply provide object structure and business rules that should be applied to data as an enterprise message object is being populated.

To use EO documents in a different location, change the EnterpriseObjectDocument@docUri attribute in the MessageObjectConfig Elements associated to your applications object in its deployment descriptor.

Like the deployment descriptors mentioned above, these artifacts should also be stored in a version control system.

Sample deployment structure:

```
Environments/Development/EnterpriseObjects/com/any-erp-vendor/Person/1.0/BasicPersonEO.xml
Environments/Development/EnterpriseObjects/com/any-erp-vendor/Person/1.1/BasicPersonEO.xml
Environments/Development/EnterpriseObjects/com/any-erp-vendor/Person/1.0/EmergencyContactEO.xml
Environments/Development/EnterpriseObjects/com/any-erp-vendor/Resources/1.0/NameEO.xml
Environments/Development/EnterpriseObjects/org/openeai/Resources/1.0/ControlAreaEO.xml
Environments/Development/EnterpriseObjects/org/any-openeai-enterprise/Person/1.0/BasicPersonEO.xml
```

It is important to note the use of the reverse domain in the deployment structure listed here. This ensures uniqueness: no two objects with the same name will collide with each other when deployed. For example, if you and your ERP both have BasicPerson objects included in your respective MOAs, you must avoid deploying two BasicPersonEO.xml files which collide. Using the reverse domain eliminates this issue and is a common development practice that will be familiar to Java developers.

Also, it is important to note the use of a version number in the deployment structure. This is so multiple versions of an object can be supported. When an MOA is generated along with these EO documents, it is generated at a particular version. Object definitions may change as time goes on. When that happens, a new version of the object is also generated. Applications can then be written to support one, both, or many versions of the same object. These version numbers correspond directly to the version of the message object definition.

Finally, you must be aware that higher-level objects like BasicPerson, EmergencyContact etc. will “include” other lower level child objects such as Name, Address, etc. The higher-level objects are objects that perform actions (query, create, update, delete or generate) and the child objects are part of them. An EO document contains the entire definition of an object. However, since we generally don’t want to have an object defined twice if it’s used in more than one higher-level object, we break these child objects into EO documents and include them in the higher-level object at runtime, so you have the entire definition of the object. In the same way as
we break Java objects and the message object definitions themselves into smaller pieces. This is all intended to fulfill an important goal of object oriented development, object reuse. However, in this case as well as in the case of our message definitions and Java objects, we’re reusing actual business object definitions. With the use of EO documents, we’re also reusing the data rules that should be applied to those business objects. This is good.

Here’s an example. The BasicPerson message object definition says it contains a Name Element as well as zero or more Address Elements. The EO document for BasicPerson therefore includes the NameEO.xml and AddressEO.xml files to make its definition complete. EmergencyContact also contains Name and Address Elements, and also does not include its own definitions of those child objects, it includes the same NameEO.xml and AddressEO.xml files that are included by the BasicPerson object. This way they both use the same definition of Name and Address objects. If there needs to be special formatting rules applied to the EmergencyContact Name object that don’t exist in the common Name, that definition can be specified in the EmergencyContactEO.xml file or in a completely different EO document, like MyDepartmentNameEO.xml, but this should be a rare occurrence.

The ControlAreaEO.xml file is special. It is a required part of the OpenEAI Message Protocol. The OpenEAI APIs use this file to build the ControlAreaRequest, ControlAreaSync or ControlAreaReply portion of an enterprise message from the contents of an enterprise Java object when an action is performed on that object.

This file is packaged with the OpenEAI distribution and on the OpenEAI website. It can be deployed along with all the other EO documents that make up your MOA but it should be copied from the OpenEAI distribution or website. It should not be changed!!!

**Analysis By-Products (Primed Documents, DTDs, and Message Definitions)**

Sample XML documents are a by-product of the OpenEAI Methodology and analysis process. The OpenEAI foundational APIs leverage these by-products. “Primed documents” are used by the foundation when an action is performed on a Java message object and that object creates the appropriate enterprise message. When an enterprise message is created, the DataArea portion of the message is an XML representation of the data contained in the Java message object. However, the object really doesn’t know everything about the rest of the OpenEAI Message Protocol. Therefore, we use the sample XML documents that are a by-product of the methodology to give the objects the structure they need to create the entire message. Dynamic portions of the documents are filled in by the message object at when the action is performed.

Using these primed documents improves the speed of building XML message documents. These documents are specified when an object is configured and used when the action is performed on the object. The only primed documents that should be specified for an object are the message documents that the object will actually need, based on the actions that may potentially be performed. For example, if an application needs to send com.any-erp-vendor.Person.BasicPerson.Query-Request messages, then that BasicPerson Java object needs to have the appropriate com.any-erp-vendor.Person.BasicPerson.Query-Request.xml primed document specified in its MessageObjectConfig section in the application’s deployment descriptor. If that is the only message action that may be performed with that message object, this is the only primed document that needs to be listed. If the application will also be creating BasicPerson objects, it will need the com.any-erp-vendor.Person.BasicPerson.Create-Request.xml primed document associated with it as well. This also proves useful in documenting
the application being developed. By looking at the message objects used and the actions they’re allowed to perform in the deployment descriptor, you can glean a lot of useful information about the application itself. See the OpenEAI API Introduction Document for more information regarding how these objects use primed documents.

As with the other artifacts, primed message documents can be posted in various places. They can be used from a web server and they can be used from a local file system. This is all specified with a URI in the deployment descriptor associated with the application using the Java message object

OpenEAI recommends that these artifacts be deployed on a web server so you don’t have to replicate them everywhere you want to run applications or gateways. Additionally, these sample messages that serve as primed documents also can serve the dual purpose of example messages which humans can read. The OpenEAI Methodology Document outlines how these artifacts can be used to complement messaging application and gateway documentation if they are posted on a web server.

Another by-product of the analysis process are the actual message definitions. These are the official specifications of the structure of messages and message objects. These definitions are determined during the analysis process and are realized in a group of files collectively referred to as “Resources”. The resources files are DTDs that contain the actual definition of each enterprise message object an organization needs. There are typically four files involved:

- **Resources.dtd** — Includes all resources files from external organizations that an organization may wish to use in their message definitions as well as their own domains, fields, and segments.
- **Domains.dtd** — Contains definitions for special data types used by simple and complex elements.
- **Fields.dtd** — Contains all simple element definitions that are children of some complex element.
- **Segments.dtd** — Contains all complex element definitions for the organization. This file, along with the other analysis by-products (DTDs and sample XML files) are then used to generate an organization’s MOA (Java objects and EO documents).

These files are included in an enterprise message definition (DTD) for a particular object. This way, we don’t have to fully define an object in all the enterprise messages (DTDs) that use that object. We only have to specify that definition once in these files, then include them into the DTD we’re putting together.

For example, the com.any-erp-vendor.Person.BasicPerson message object has several actions associated with it: create, update, delete and query. Each of these actions has a corresponding DTD associated with it that defines what an enterprise message for that object and action looks like. There’s:

- com.any-erp-vendor.Person.BasicPerson.Create-Request.dtd
Each of these enterprise messages contain in their DataArea a com.any-erp-vendor.Person.BasicPerson enterprise message object. By including the Resource files mentioned above for the any-erp-vendor.com company, which includes the definition of what a BasicPerson message object is, these files don’t have to provide that definition. All they have to do is say there’s a com.any-erp-vendor.Person.BasicPerson message object in the DataArea of each document.

Refer to the OpenEAI Message Definitions document for more information on these resource files and how they’re used.

The resource files are included in the enterprise message definitions (DTD) relatively, so it doesn’t matter where the resource files are stored, as long as they’re stored according to the pattern they’re included. For example, “four levels down under the Resources/1.0 tree” or “../../../Resources/1.0.”

Sample Deployment Structure for sample documents and DTDs (the by-products of the analysis process):

Message/releases/com/any-erp-vendor/Person/BasicPerson/1.0/xml/Query-Request.xml
Message/releases/com/any-erp-vendor/Person/BasicPerson/1.0/dtd/Query-Request.dtd
Message/releases/com/any-erp-vendor/Person/BasicPerson/1.1/xml/Query-Request.xml
Message/releases/com/any-erp-vendor/Person/BasicPerson/1.1/dtd/Query-Request.dtd

Message/releases/com/any-erp-vendor/Employee/BasicEmployee/1.0/xml/Query-Request.xml
Message/releases/com/any-erp-vendor/Employee/BasicEmployee/1.0/dtd/Query-Request.dtd
Message/releases/com/any-erp-vendor/Employee/BasicEmployee/1.1/xml/Query-Request.xml
Message/releases/com/any-erp-vendor/Employee/BasicEmployee/1.1/dtd/Query-Request.dtd

Message/releases/com/any-erp-vendor/Resources/1.0/Segments.dtd
Message/releases/com/any-erp-vendor/Resources/1.0/Domains.dtd
Message/releases/com/any-erp-vendor/Resources/1.0/Fields.dtd
Message/releases/com/any-erp-vendor/Resources/1.0/Resources.dtd

**Runtime scripts**

OpenEAI supplies several runtime management scripts in its distribution which can be used for starting and stopping gateways and applications, rotating the logs associated with a running gateway or application, and setting up a working shell environment for administering gateways and applications. While it is certainly not required that you use these scripts, it is recommended. They will simplify the administration of these components.

*It is important to note that these scripts assume you have followed the other recommended deployment patterns within this document. Specifically, regarding user accounts and directory structures of the deployed environment. If you choose to deploy applications and gateways in another way, these scripts may not work. Therefore, you will*
either need to change these scripts appropriately or use another mechanism for actually running and administering your applications.

Currently, there are scripts available for the Unix/Linux environment only. As you'll see below, OpenEAI recommends that gateways running on the Windows platform be deployed as services so these scripts are not needed in that environment. The following scripts are supplied in the OpenEAI distribution and should be deployed into an environment:

start

This script first checks for a start script in the current directory from which the script is invoked, and if present, invokes it. This is to allow an arbitrary library directory (LIB_DIR) parameter and to allow administrators to override other default start script behavior with their own custom start script if they wish.

This script also expects one and only one empty file used for tagging purposes named api-release-[name] to exist in the deployment directory, where [name] is the name of the JAR set this application or gateway should use. As discussed earlier in this document, the need for different JAR sets can arise for a number of reasons. Most frequently, the OpenEAI project has used different constellations of JAR files with different releases of the OpenEAI APIs and its dependent APIs, so we have used JAR sets like 1.0, 2.0, 2.5, and 3.0. However, you could just as easily have JAR sets for any other purpose and named SALLY or BOB as well. So tag file names may end up looking like api-release-2.0, api-release-3.0, api-release-SALLY.

If a properties file is specified as the optional second parameter, it is used. Otherwise the .properties file in the directory from which the start script was invoked is used.

USAGE: start [type] (properties)
        type: (consumer | application)
        properties: optional if properties file exists in the directory from which start is invoked

The start script calls lower level scripts (appctl and consumerctl) depending on the type of component being started. It requires one parameter, the type of component being started (a "consumer" or an "application"). Starting applications and gateways with the start invokes appctl or consumerctl which create a process ID file (pidfile) that contains the process ID of the gateway or application started.

stop

This script kills the process listed in the pidfile present in the gateway or application deployment directory. This pidfile is created by appctl and consumerctl when an application or gateway is started.

USAGE: stop [type]
        type: (consumer | application)

appctl

This script is used to start any OpenEAI scheduled application. These are applications that are all started with the org.openeai.afa.GenericAppRunner class. See the OpenEAI API Introduction Document for details on scheduled applications.
The start script (listed above) passes the properties and library directory parameter to this script. Then appctl builds the appropriate classpath based on the JAR set tag that exists in that directory. Then it instantiates the `org.openeai.afa.GenericAppRunner` class passing the properties file name it was passed to that application. Finally, the `GenericAppRunner` class reads the properties file, and builds the appropriate `AppConfig` which loads the deployment descriptor and instantiates a `ScheduledApp` and one or more `ScheduledCommands`. Once that is done, the application behaves as prescribed by the configuration information found in the deployment descriptor (the Schedule).

**consumerctl**

The consumerctl script behaves almost identically to the appctl script (listed above). However, instead of starting the `GenericAppRunner` class, it invokes the `org.openeai.jms.consumer.MessageConsuerClient` class, passing the location of the properties file to it. Then the `MessageConsumerClient` class builds an `AppConfig` object which loads the appropriate deployment descriptor referenced in the properties file and instantiates the Consumers and ConsumerCommands associated to this gateway. Then when any consumer consumes a message, it executes the appropriate command.

**restart**

Restart performs a stop and then a start of the gateway or application and has the same usage as the start script.

**rotatelog**

This script rotates the logfile for applications and gateways. When application or gateways are started, their logging output is redirected to a file in their deployment directory call application.log or consumer.log. Frequently, you will want to rotate these logs to clear out previous log messages and save them in a more permanent, compressed state. Executing rotatelog copies the current contents of the logfile into a file within the directory ./archived-logs in a file named to reflect the exact time rotatelog was executed. Then rotatelog uses gzip to compress the archived log file and clears the logfile in the deployment directory, so you have a clean, working logfile.

**USAGE:** rotatelog [type]

  type: (consumer | application)

**setenv**

This script sets various environment variables according to the environment passed in. For example, `. setenv Development` will set the base messaging environment for all subsequent commands executed to be that of the Development environment. This allows you to deploy the artifacts by environment and switch between them at runtime. This script should be in the path of the messaging administration user account that is used to administer messaging gateways and applications.

**USAGE:** . setenv [Environment Name]

**Runtime administration and monitoring**
Once a gateway or application has been deployed and started, it should be monitored. This includes monitoring the status of the running process and, in the case of gateways that handle request message, monitoring that consumer’s ability to reply to requests.

There are many tools available to help with the first requirement. Process level monitoring is not a new concept and products are available for both Unix/Linux platforms as well as Windows platforms.

For the Windows platform, we have used a product called Servers Alive which is manufactured by Woodstone. This product has worked well for the existing Windows based OpenEAI deployments but there are other products available that would most likely work as well. We just haven’t researched or used them.

OpenEAI does provide a mechanism for process-level monitoring for the Unix/Linux platform. This facility is called “monitor.” It consists of a script called monitor and a configuration file called monitor.config. These two artifacts are used to specify and monitor components that should be monitored at a process level. The monitor script should be scheduled using cron or some other standard scheduling facility to run at an interval such as every minute or whatever is appropriate for your organization. When scheduled, monitor will run at the specified interval and use the monitor.config file to monitor the processes on the current host as indicated in monitor.config and take appropriate action if it detects that processes that are supposed to be running are not running.

The appctl and consumerctl scripts mentioned above, create a pidfile that contains the process id of the application or gateway. The monitor script is directed by monitor.config to keep an eye on these processes and send an email to appropriate support personnel if the process should ever terminate abnormally. Below is a brief summary of the features of this monitor facility.

The monitor.config file specifies which processes should be monitored. Additionally, it specifies the level at which they should be monitored. Levels are used to instruct the monitor script how to behave should the process being monitored terminate. The levels are:

**LOW:** Instructs the monitor script to simply log the fact that the process has terminated. This is a useful level that allows routine maintenance to be performed on an application or gateway without sending alerts.

**MED:** instructs the monitor script to send an email to the specified support group should the process terminate. This gives that group a chance to respond to the fact that the process has terminated.

**HIGH:** instructs the monitor script to send an email to the specified support group. Additionally, this setting instructs the monitor script to attempt to restart the application or gateway being monitored.

**HALT:** instructs the monitor script to terminate the gateway or application.

As with everything else mentioned throughout this document, the monitor.config file should be deployed though a software configuration management or version control system. Anytime changes are made to a file and checked in to the configuration system, it should be automatically made available to all machines running these applications or gateways. Then the monitor script will re-read the file each time it runs and behave according to the latest information found in the monitor.config file. Same as with JAR files and other scripts, it is important to be able to make
changes to this file and make these changes known to the entire enterprise without having to manually move the file to all physical machines running messaging applications.

Below is an example monitor.config file for the sample OpenEAI enterprise used in the Getting Started with OpenEAI Document.

Note that there can be no spaces in any of the information specified in the monitor.config file. This means, no empty lines and no spaces between words. Therefore, the table shown below may be hard to read. It is recommended that you pull the information from that file into an editor that will allow you to turn off line wrapping so you can see the full lines with no breaks.

Here is a description of what’s actually shown in this file:


HOSTNAME = not fully qualified Hostname
ENVIRONMENT = parameter used to call setenv
TYPE = parameter used to call start
PRIORITY = LOW (display status only)
          MED (e-mail CONTACT when down)
          HIGH (e-mail and attempt to restart when down)
          HALT (Stop the process, if it is on the host where monitor is executed)
SYMBOLIC_NAME = any descriptor -- no spaces
PID-FILE-PATH = full path to the pidfile (including filename)
CONTACT = e-mail address (no spaces)

Note: you can comment out a line in this configuration file by putting a # as the first character of the line, but NO SPACES ARE ALLOWED ON ANY LINE.

```
# DEVELOPMENT_ENVIRONMENT_MONITOR_CONFIG
#
# INFRASTRUCTURE
#
#EnterpriseTransRouters
#
#EnterpriseSyncLogger
#
#EnterpriseSyncErrorLogger
```
So far, we’ve focused on process-level monitoring. This is really nothing new. However, for gateways that handle request messages, there is another, less obvious form of monitoring that should occur. The OpenEAI foundation APIs have been designed to do everything possible to
ensure that all components which require some sort of initialization and are intended to maintain a steady state with a resource are able to maintain that state, no matter what. This includes things like verifying and automatically re-establishing database connections stored in a connection pool, internally monitoring the status of a JMS producer’s connection to the broker and doing the same for JMS consumers. Through these efforts, we are able to deploy applications that run an extremely long time and usually recover from both planned and unplanned outages. This includes things such as upgrading a broker or “bouncing” a database. When events like this occur, the applications or gateways that use these resources will simply “heal” themselves when resources are available again.

However, try as we may, and do, it is certain that we will never get to the point where everything works perfectly forever.

For example, there are situations where applications or gateways are unable to detect an error or they detect an error but aren’t equipped to inform anyone that an error condition exists.

Here’s the OpenEAI solution: instead of building unrecoverable error-detecting logic into every application, we have simplified things by creating another infrastructure component, the DestinationPoller. The DestinationPoller is an OpenEAI scheduled application which is configured to poll gateways throughout the enterprise. Specifically, it sends requests to gateways and consumes their replies. If it sends a request to a gateway and receives an error in the reply, or if for some reason the request times out, it can notify the support group of the potential problem. This application is deployed in the same manner as any other application and uses all the same OpenEAI APIs to produce requests. It simply sends the same query over and over again at an appropriate interval configured by the deployers. Through its deployment descriptor it can be configured to send requests to any gateway in the enterprise. An example of the DestinationPoller is included in the Getting Started with OpenEAI Document.

**JMS Provider Administration Artifacts**

JMS Providers provide many means by which to administer their products. Therefore, it is hard to prescribe an exact structure to follow. This may include things like user and group specifications, topic and queue administration, etc. Depending on the JMS provider you use, there may not be any real “administration artifacts”.

That said, the same themes which have been mentioned throughout this document apply here as well.

- **Artifacts need to be version controlled, they need to be distinguished by environment**
- **You likely need to plan to run multiple instances of your JMS Provider in the same environment for the same reasons you run multiple instances of gateways and applications.**

It is vitally important that you have a well-planned approach to deploying any artifacts that you may use to administer these instances.

Additionally, we will recommend specific naming techniques to use when naming certain components. For example, the actual queues and topics that must be created within the broker should be named by reverse domain. For example, `org.any-openeai-`
enterprise.topic.Transrouter, or org.any-erp-vendor.queue.AevGateway. However, some products may not support these long names.

These are simple examples of some of the features that you can look for when deciding on which JMS provider to use.

The interesting thing to note is that if you decide to go with a, standards-based integration approach, these really are the type of features you have to focus on. Not what the vendor does. If they area legitimate JMS provider, you know most of what you need to know. You can simply focus on how well they do it!

Administered Objects (ConnectionFactories and Destinations)

A fundamental goal of the OpenEAI API is to be vendor-independent. Our judgments are based on and developed to specifications, not products. The APIs described above and described throughout OpenEAI documentation all revolve around many other standards-based APIs. At the broadest level, OpenEAI leverages Java and XML. The further you dig down, you’ll see that there are many other, more detailed relationships between OpenEAI and other standards based APIs. For example, OpenEAI uses JMS as a message transport. Even though OpenEAI uses JMS, it is still possible to tie it to a particular vendor. This would happen if the OpenEAI APIs leveraged any proprietary feature for a particular JMS provider. When using JMS, one of the easiest ways to tie yourself to a particular JMS provider would be to connect to that broker via a proprietary connection mechanism. However, JMS provides a facility that allows you to avoid this. By using JMS Administered Objects (ConnectionFactories and Destinations) to create all necessary objects and to connect to the broker itself, you drastically reduce the likelihood of vendor dependence. Additionally, by using JNDI to retrieve those objects, you’re isolating yourself from dependence on specific directory service implementations. No matter which JMS provider you choose, they must provide a mechanism to access administered objects via JNDI or they are not fully implementing the JMS specification. The OpenEAI APIs are continuously tested against several different JMS providers to ensure this independence.

OpenEAI recommends that these administered objects be stored in a directory server to ensure consistency and more importantly to provide the necessary security to access these objects. However, that is not a requirement. As long as the JMS provider allows a JNDI interface to those administered objects, it’s simply a matter of changing the configuration information related to an OpenEAI JMS foundation component (producers and consumers) to retrieve those objects.

Below is an example of the OpenEAI recommended deployment pattern for JMS Administered objects in a directory server:
Not only does this tree allow for multiple JMS providers, it allows for multiple versions of a particular provider. Since the provider specifies what information is stored in these administered objects, they can change from release to release of a product. Also notice that there is an environment level to the tree that allows the administered objects to be different by environment, because you will typically have different JMS provider (broker) instances for each environment. Typically, the JMS provider will provide a facility for placing these objects in the directory server. We recommend you do that once and then simply copy those objects to the appropriate places in the directory server as needed. A tool like LDAPBrowser (listed above) or the directory server console can be used to administer these objects. Remember, these are simply the administered objects that will be used by an application to connect to a particular broker, topic and queues. The actual queues and topics as well as user permissions etc. must still be created in the broker itself.

Within an environment, there is a PointToPoint and PubSub tree. All administered objects related to PointToPoint messaging (QueueConnectionFactory and Queues) are stored in the PointToPoint tree. All administered objects related to PubSub messaging (TopicConnectionFactory and Topics) are stored in the PubSub tree. These objects will contain information specific to the provider being used that allow producers and consumers to connect to the appropriate broker, topic and queues. These objects make up the artifacts actually used by the OpenEAI foundation APIs (PointToPoint and PubSub producers and consumers). They contain references to the actual implementations supplied by the JMS provider which are used at runtime. This is one way vendor independent JMS development is achieved.
CVS (Concurrent Versions System)

Several times throughout this document, versioning systems have been referred to as a means of storing and maintaining most of the artifacts related to OpenEAI application development and deployment. The OpenEAI project uses the Concurrent Versions System (CVS) for both version control of source code and as deployment management infrastructure. For more information about CVS please visit http://www.cvshome.org. While it is not required that you use CVS, it is a cheap, effective and commonly-used tool that addresses requirements that we feel are very important.

Not only is this an important requirement for all application development, as we’ll discuss, OpenEAI recommends that the artifacts stored in the version control repository be made available to all environments that will be used for application deployment. Therefore, not only do we use CVS as a versioning system, we also use it to help organize as well as actually deploy our runtime environments. The artifacts associated with each environment (jar files, scripts, deployment descriptors, broker configuration files and scripts, EO documents etc.) are all deployed using tools available in the CVS suite.

How do you deploy to many servers through CVS? There are probably many ways to do this, but here are two ways the OpenEAI Project and some OpenEAI sites that use CVS have done it.

Synchronize a Checked-out Tree

CVS provides a mechanism to maintain a checked-out tree or a tree with the current version of every artifact in the CVS repository or in a specific module or modules of the repository. Through the CVS run command interface, you can keep this checked out tree current. Basically, when anybody checks in a version of a deployment artifact, the checked-out tree of these configuration artifacts is updated. Appropriate portions of this checked-out tree can then be synchronized (using rsync or other utilities) to file systems on all of your server to which you deploy servlets, gateways, application, message brokers, web server, and just about anything else.

Export the Checked-out Tree via NFS

This approach is the same as above, but instead of actually synchronizing the deployment artifacts to other servers, appropriate portions of the checked-out tree are merely exported via NFS and mounted to the appropriate servers. The upside to this approach is that there can be literally no latency between the time you check a new configuration into CVS and the time it is available to gateway or application servers. However, the downside is that NFS can be cumbersome to work with, unreliable on some networks, and if you lose your versioning system, hardware you need to have a standby ready to export and mount to all those servers, because the deployment artifacts really only reside on your versioning server.

Gateways and applications

Now that we’ve discussed the supporting artifacts, we’ll discuss gateways and applications developed using the OpenEAI APIs. OpenEAI gateways and applications are simply Java applications that implement a pattern of execution and utilize other OpenEAI artifacts to configure themselves. To learn more about the patterns of execution as well as the artifacts used by OpenEAI applications, please refer to the OpenEAI API Introduction Document. These applications must be deployed in such a way as to meet the goals mentioned above.
For message gateways in particular, it is important to deploy multiple instances of that gateway in order to achieve things like fault tolerance and scalability. Regardless of your JMS provider, this is an achievable goal. Not only should multiple instances of a gateway be deployed, but ideally, that gateway should be deployed on different hardware to avoid issues caused by hardware failure. If a gateway supports request/reply messaging, it is very simple to have multiple consumers connected to the same JMS Queue. The first consumer that consumes the message, destructively consumes that message and no other consumer will get it. For gateways that support Publish-Subscribe (synchronization) messaging, it is a little different. All PubSubConsumers will consume a message delivered to a given topic. However, the OpenEAI APIs provide a mechanism that determines if any other consumer has consumed that message and if so, ignores it.

For OpenEAI applications, it may be a case-by-case basis as to whether or not there needs to be multiple instances of a particular application. Most times, it is desirable to deploy multiple instances of a servlet to achieve the same goals mentioned above.

Next, we will discuss actual gateway deployment to demonstrate these recommended patterns.

Each deployed gateway that uses the same Deployment document is considered an “instance” of that gateway. For example, in our fictitious organization Any-OpenEAI-Enterprise.org, there needs to be several instances of the OpenEAI reference implementation EnterpriseRequestProxy. This is simply a gateway that proxies requests between an un-trusted non-authoritative application and an authoritative system. See the OpenEAI Implementation Strategies Document document for more information on this particular gateway and its purpose. The reason for multiple instances is so that in case one instances goes down abnormally or needs to be taken down for hardware maintenance etc. the organization will still have an instance up handling all the requests supported by the proxy gateway. There is only one Deployment document associated to this gateway. Each instance of the gateway is deployed onto different machines or at least into different structures on the same machine. Then, any single instance can be brought down and all messaging activity will go through the remaining instance. All the running instances of this gateway can be referred to as a gateway cluster. That is, a cluster of EnterpriseRequestProxy instances.

Note, these are the manual instructions for deploying an OpenEAI environment following the recommended patterns. This can also be accomplished by running the appropriate OpenEAI environment scripts to establish these structures. These scripts will be made available on the OpenEAI website.

OpenEAI recommends the following pattern for deploying gateways and applications:

**Deployment on Unix/Linux**

1. Create a user called openeai with a home directory of `/opt/openeai`. By doing this, the openeai user will own everything underneath and will have the appropriate permissions to execute and maintain the environment.

2. Create a directory called `/opt/openeai/messaging`

3. Create the appropriate environment directories under `/opt/openeai/messaging`. For example:
4. Create an applications and gateways directory under each environment. This is where you’ll locate instances of these components. For example, under the gateways directory we might have something that looks like this:

/opt/openeai/messaging/Development/gateways/AevGateway/gw101-1

where gw101 is the machine name and 1 is the instance of the gateway. This way, we can identify which system process is associated to the running gateway on the server. This is where the properties file associated to this gateway will be deployed. Note, this is the only artifact that you must deploy with the application that isn’t automatically available via the NFS mount. If you want to deploy multiple instances of this gateway on the same (or different) machine(s), you simply put this properties file in the appropriate directory for the other instance you wish to deploy. This is very cool.

5. Create a symlink for the jars directory to the NFS mounted checked out CVS tree. For example:

a. ln –s /cvs/configs/Environments/Development/Jars jars

This will give you access to all the runtime libraries in the Development environment. Then when you start an application or gateway, you can create the “api-release-x.x” file to tell the start script which version of the jar files to use within the Development environment.

6. Create a symlink for the scripts directory to the NFS mounted checked out CVS tree. For example:

a. ln –s /cvs/configs/Environments/Development/Scripts scripts

This will give you access to all the Development environment scripts that will be used to manage the gateways and applications (appctl, consumerctl, start, stop etc.)

### Deployment on Windows

Because of the location independent deployment patterns recommended by OpenEAI, there is really very little difference between the deployment of applications and gateways on the Windows platform compared to a Unix or Linux deployment. The runtime scripts provided by OpenEAI are only available for the Unix and Linux platform. However, when running OpenEAI based applications or gateways on Windows, it is recommended that you deploy them as services using a product like the Java Service Launcher. This is to allow the application or gateway to run without a user being logged into the system. Clearly, these applications can be run as simple Java applications following a similar pattern as described on the Unix or Linux platform but running them as services allows the applications or gateways to continue executing even when no one is logged in. Additionally, it makes it easy to start the applications or gateways when the system is first started.
To run the applications or gateways on Windows, you must make the same artifacts available to the JRE when you start the applications or gateways. More specifically, you must make the runtime libraries (JAR files) available when you start them. The only other artifact that must be deployed is the properties file associated to that gateway or application. If you follow the recommended deployment patterns for the rest of the artifacts needed at runtime, you will be able to access them from any machine you wish to run and OpenEAI based application or gateway.

To install an OpenEAI based application or gateway as a service on Windows (NT/2000/XP):

1. Create the appropriate directory structure where the applications will be run. For example:
   `c:\OpenEAI\Messaging\Development\applications\AevApplication\gw101-1` or `c:\OpenEAI\Messaging\Development\gateways\AevGateway\gw101-2` etc.

2. Place the appropriate properties file in the above location for the application or gateway you’re deploying.

3. Make the JAR files available and add them to the classpath. The JAR files can either be simply copied to the appropriate local file system or they can be made available via a network drive that is available to the machine running the application or gateway. If the files are copied, they must be recopied anytime any of them change. This is one reason to make them available via a network drive that is automatically refreshed when the libraries are checked into your versioning system.

   The libraries can either be made available system level (setting the CLASSPATH environment variable in the System environment) or at runtime by using the `-classpath` option with the Java command used to start the application or gateway.

4. Run either the `org.openeai.jms.consumer.MessageConsumerClient` or the `org.openeai.afa.GenericAppRunner` passing the name of the properties file associated to the application or gateway your starting and providing the appropriate classpath to the Java command. Depending on the JMS provider you’re using, there could be additional environment settings specified along with the java execution.

   There will be varying methods of actually invoking the JRE depending on the product you use to start this application or gateway. Additionally, there will be a step that isn’t listed here which is, installing the application or gateway as a service. The products used to do this will have facilities included with them that allow you to do this. The Java Service Launcher allows you to specify all this information in a .ini file that looks something like this:

   ```ini
   [service]
   appname = AevGateway Development gw101-1
   servicename = AevGateway-Development-gw01-1
   displayname = AevGateway-Development-gw101-1
   ;port used to communicate stop command to JVM
   ;Must be unique for each JSL instance installed
   stopport = 8465
   ;service start type
   ```

   Note: This example does not include all the information that can be specified with this particular product but it does include sufficient information for the gateway we’re starting.
;auto demand disabled
default is auto
starttype=auto

;load ordering group
loadordergroup=someorder

;Allocate a console and register a console event handler to catch shutdown events.
;Default is true; options are FALSE TRUE false true YES NO yes no
;This option has two effects:
;1. If active it catches the logoff events and prevents the service from erroneously stopping in this case.
;2. It creates a console window if interaction with the desktop is allowed.
;Effect 1 can be achieved in Java 1.3.1 and higher by using the -Xrs option.
;Effect 2 may or may not be desired.
useconsolehandler=true

;Call <stopclass>.<stopmethod> through JNI so stop the JVM.
;This can be used an alternative to the ServiceStopper
;When all three parameters are defined the JSL will try to use JNI to stop the service.
;If not it will use a tcp/ip connection to the stopport.
;The simplest way to use this functionality is to use the System.exit() call as specified below.
;Nevertheless you can call any static method in your code.
;The method called is expected to terminate the JVM one way or the other. It can directly
;call System.exit() or make another Thread do it make the main method return.
;The method can return immediately if desired. If the JVM doesn't stop another attempt can be made to
;stop the service at a later time.

stopclass=java/lang/System
desiredly=exit

;Take care to specify the right signature for your method. void System.exit( int code ) has
;the signature specified below. void myMethod() has the signature ()V. Please refer to the
;JNI documentation for details.
stopsignature=(I)V

[java]
;Path to the java runtime used
;If this option is not used the default from the registry will be used
jrepath=C:\Program Files\JavaSoft\JRE\1.3.1_02

;working directory
wrkdir=C:\openeai\messaging\Development\gateways\AevGateway\gw101-1

;the number of parameters passed to the java application
;parameters must be numbered param00, param01 ...
params = 8
param00 = -Xms128m
param01 = -Xmx256m
param02 = -DSSL_CA_CERTIFICATES_DIR=NONE
param03 = -DSonicMQ.TCP_NODELAY=TRUE
param04 = -cp
param05 =
param06 = org.openeai.jms.consumer.MessageConsumerClient
param07 = C:\openeai\messaging\Development\gateways\AevGateway\gw101-1\AevGateway.properties
Appendix 1: The GNU Free Documentation License

GNU Free Documentation License
Version 1.1, March 2000

Copyright (C) 2000 Free Software Foundation, Inc.
59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other
written document "free" in the sense of freedom: to assure everyone
the effective freedom to copy and redistribute it, with or without
modifying it, either commercially or noncommercially. Secondarily,
this License preserves for the author and publisher a way to get
credit for their work, while not being considered responsible for
modifications made by others.

This License is a kind of "copyleft", which means that derivative
works of the document must themselves be free in the same sense. It
complements the GNU General Public License, which is a copyleft
license designed for free software.

We have designed this License in order to use it for manuals for free
software, because free software needs free documentation: a free
program should come with manuals providing the same freedoms that the
software does. But this License is not limited to software manuals;
it can be used for any textual work, regardless of subject matter or
whether it is published as a printed book. We recommend this License
principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work that contains a
notice placed by the copyright holder saying it can be distributed
under the terms of this License. The "Document", below, refers to any
such manual or work. Any member of the public is a licensee, and is
addressed as "you".

A "Modified Version" of the Document means any work containing the
Document or a portion of it, either copied verbatim, or with
modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of
the Document that deals exclusively with the relationship of the
publishers or authors of the Document to the Document's overall subject
(or to related matters) and contains nothing that could fall directly
within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further
copying of the copies you make or distribute. However, you may accept
compensation in exchange for copies. If you distribute a large enough
number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and
you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies of the Document numbering more than 100,
and the Document's license notice requires Cover Texts, you must enclose
the copies in covers that carry, clearly and legibly, all these Cover
Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on
the back cover. Both covers must also clearly and legibly identify
you as the publisher of these copies. The front cover must present
the full title with all words of the title equally prominent and
visible. You may add other material on the covers in addition.
Copying with changes limited to the covers, as long as they preserve
the title of the Document and satisfy these conditions, can be treated
as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit
legibly, you should put the first ones listed (as many as fit
reasonably) on the actual cover, and continue the rest onto adjacent
pages.

If you publish or distribute Opaque copies of the Document numbering
more than 100, you must either include a machine-readable Transparent
copy along with each Opaque copy, or state in or with each Opaque copy
a publicly-accessible computer-network location containing a complete
Transparent copy of the Document, free of added material, which the
general network-using public has access to download anonymously at no
charge using public-standard network protocols. If you use the latter
option, you must take reasonably prudent steps, when you begin
distribution of Opaque copies in quantity, to ensure that this
Transparent copy will remain thus accessible at the stated location
until at least one year after the last time you distribute an Opaque
copy (directly or through your agents or retailers) of that edition to
the public.

It is requested, but not required, that you contact the authors of the
Document well before redistributing any large number of copies, to give
them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under
the conditions of sections 2 and 3 above, provided that you release
the Modified Version under precisely this License, with the Modified
Version filling the role of the Document, thus licensing distribution
and modification of the Modified Version to whoever possesses a copy
of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct
   from that of the Document, and from those of previous versions
   (which should, if there were any, be listed in the History section
   of the Document). You may use the same title as a previous version
   if the original publisher of that version gives permission.
B. List on the Title Page, as authors, one or more persons or entities
   responsible for authorship of the modifications in the Modified
   Version, together with at least five of the principal authors of the
   Document (all of its principal authors, if it has less than five).
C. State on the Title page the name of the publisher of the
   Modified Version, as the publisher.
D. Preserve all the copyright notices of the Document.
E. Add an appropriate copyright notice for your modifications
   adjacent to the other copyright notices.
F. Include, immediately after the copyright notices, a license notice
   giving the public permission to use the Modified Version under the
   terms of this License, in the form shown in the Addendum below.
G. Preserve in that license notice the full lists of Invariant Sections
   and required Cover Texts given in the Document's license notice.
H. Include an unaltered copy of this License.
I. Preserve the section entitled "History", and its title, and add to
   it an item stating at least the title, year, new authors, and
   publisher of the Modified Version as given on the Title Page. If
   there is no section entitled "History" in the Document, create one
   stating the title, year, authors, and publisher of the Document as
   given on its Title Page, then add an item describing the Modified
   Version as stated in the previous sentence.
J. Preserve the network location, if any, given in the Document for
   public access to a Transparent copy of the Document, and likewise
   the network locations given in the Document for previous versions
   it was based on. These may be placed in the "History" section. You
   may omit a network location for a work that was published at
   least four years before the Document itself, or if the original
   publisher of the version it refers to gives permission.
K. In any section entitled "Acknowledgements" or "Dedications",
   preserve the section's title, and preserve in the section all the
   substance and tone of each of the contributor acknowledgements
   and/or dedications given therein.
L. Preserve all the Invariant Sections of the Document,
   unaltered in their text and in their titles. Section numbers
   or the equivalent are not considered part of the section titles.
M. Delete any section entitled "Endorsements". Such a section
   may not be included in the Modified Version.
N. Do not retile any existing section as "Endorsements"
   or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or
appendices that qualify as Secondary Sections and contain no material
 copied from the Document, you may at your option designate some or all
 of these sections as invariant. To do this, add their titles to the
 list of Invariant Sections in the Modified Version's license notice.
These titles must be distinct from any other section titles.

You may add a section entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties—for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled "History" in the various original documents, forming one section entitled "History"; likewise combine any sections entitled "Acknowledgements", and any sections entitled "Dedications". You must delete all sections entitled "Endorsements."

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in
the collection, provided that you follow the rules of this License for
verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute
it individually under this License, provided you insert a copy of this
License into the extracted document, and follow this License in all
other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate
and independent documents or works, in or on a volume of a storage or
distribution medium, does not as a whole count as a Modified Version
of the Document, provided no compilation copyright is claimed for the
compilation. Such a compilation is called an "aggregate", and this
License does not apply to the other self-contained works thus compiled
with the Document, on account of their being thus compiled, if they
are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these
copies of the Document, then if the Document is less than one quarter
of the entire aggregate, the Document's Cover Texts may be placed on
covers that surround only the Document within the aggregate.
Otherwise they must appear on covers around the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may
distribute translations of the Document under the terms of section 4.
Replacing Invariant Sections with translations requires special
permission from their copyright holders, but you may include
translations of some or all Invariant Sections in addition to the
original versions of these Invariant Sections. You may include a
translation of this License provided that you also include the
original English version of this License. In case of a disagreement
between the translation and the original English version of this
License, the original English version will prevail.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except
as expressly provided for under this License. Any other attempt to
copy, modify, sublicense or distribute the Document is void, and will
automatically terminate your rights under this License. However,
parties who have received copies, or rights, from you under this
License will not have their licenses terminated so long as such
parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE
The Free Software Foundation may publish new, revised versions
of the GNU Free Documentation License from time to time. Such new
versions will be similar in spirit to the present version, but may
differ in detail to address new problems or concerns. See

Each version of the License is given a distinguishing version number.
If the Document specifies that a particular numbered version of this
License "or any later version" applies to it, you have the option of
following the terms and conditions either of that specified version or
of any later version that has been published (not as a draft) by the
Free Software Foundation. If the Document does not specify a version
number of this License, you may choose any version ever published (not
as a draft) by the Free Software Foundation.