



xTier™ Micro Kernel

The xTier™ Micro Kernel provides flexible and simple integration with a wide variety of deployment environments.

Key Benefits:

- Removes dependencies between xTier™ and the underlying deployment environment.
- J2SE (in-process), JBoss 3.x/4.x and WebLogic 7.x/8.x micro kernels provided out-of-the-box.
- Custom micro kernels are easily implemented for custom deployment environments
- Allows application developed with xTier™ to be easily deployed and then re-deployed in a different deployment environment

Introduction - xTier™ Micro Kernel-based Integration

The term “micro kernel” has been used in the computer industry for a long time and can be applied to a variety of different applications and technologies, from a processor’s micro kernel through to a micro kernel as a joint module of the program’s components.

History of the Term “Micro Kernel”

The concept of a micro kernel was introduced by the Next company, in their operating system (OS) which used the Mach micro kernel developed by the University of Utah. The small privileged kernel of this OS, surrounded by subsystems executed in user-mode, provided a highly-flexible and modular system (although in practice this advantage was slightly devaluated by Next’s choice to use a server implementing UNIX BSD 4.3 OS as the shell for the Mach micro kernel).

The next micro kernel-based operating system was Microsoft Windows NT, in which the key advantage of the micro kernel was not only modularity, but also portability. (It should be noted that there is no common agreement on whether NT is actually a micro kernel-based OS or not.) As the NT environment was required to execute DOS, Windows, OS/2 and Posix compatible programs Microsoft chose to use the micro kernel design to provide modularity so that the NT kernel for NT’s common structure was separated from the subsystems which implemented each of these execution environments.

Later operating system micro kernel architectures were declared by companies like Novell/USL, Open Software Foundation (OSF), IBM, Apple, Sun (SpringOS), QNX Software Systems, Unisys and others.

The Microkernel Approach in Software Applications

Although the terms "microkernel" and "microkernel technology" are generally used within the context of operating systems, the terms do have a wider application. Although there are other terms rather than "micro kernel" which may be applied to the notions discussed in this whitepaper, the term itself is of less importance than the underlying concepts, which remain the same, regardless of nomenclature.

The basic idea incorporated by the term micro kernel, whether it is applied to operating systems or other software, is to design a small, lightweight environment specific interface module to insulate the core technology from the differences in specific deployment environments. The kernel and the micro kernel have a well defined "contract" with the micro kernel providing a specific set of functions to the kernel, and thus insulating the kernel from the underlying deployment environment. This is the approach used by xTier™ and accounts for the ease with which xTier™ can be deployed in a variety of environments as well as providing the ability to create micro kernels for custom applications.

The advantages of this approach are:

- uniform interface
- expandability
- portability
- reliability
- support for an object oriented approach

Without using a micro kernel the kernel would have to be modified every time a new deployment environment was to be supported. This, understandably, increases the complexity of creating, testing and maintaining the core. Further, providing the facility to allow custom deployments is problematic for the kernel developer, as without a micro kernel design, direct modification of the kernel to support different deployment environments would be required. This is by no means a trivial task, and is one that requires a thorough understanding of the kernel itself. At best, erroneous modification of the kernel would result in the loss of system serviceability and at worst, create a completely unstable and unusable system.

As such, it clearly makes sense to provide a micro kernel interface, (which provides the abstraction of the minimal functionality required by the kernel), which can then be implemented in a variety of hosting environments without direct modification to the kernel itself. Furthermore, without using a micro kernel the ability to deploy xTier™ on a variety of systems would be limited to those directly supported by the kernel.

The xTier™ Micro kernels

Although the functionality of most xTier™ services does not depend on the specific execution environment it is still necessary for the xTier™ kernel to have access to the characteristics of the underlying environment. The xTier™ installation set includes micro kernels for several common deployment environments, (J2SE in-process, JBoss 3.x/4.x and WebLogic 7.x/8.x). The developer can supplement the pre-build micro kernels to support custom deployment environments which take into account the specific features and requirements of these environments. In addition, xTier™ provides the source code for the J2SE in-process micro kernel as well as the JBoss 3.x micro kernel to provide the developer with a detailed example for the basis of their custom implementations.

xTier™ Micro Kernel Tasks

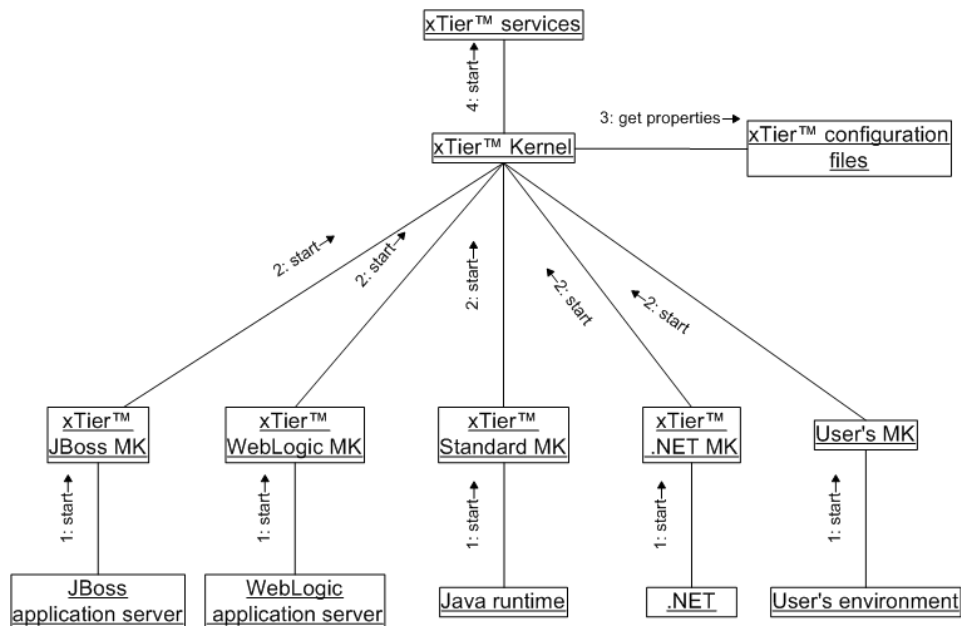
The xTier™ micro kernel performs the following tasks:

- start the xTier™ Kernel in the particular execution environment
- pass the execution environment context to xTier™ as an argument to the xTier™ Kernel method **start(...)**
- shutdown xTier™ – the micro kernel should provide an opportunity to perform certain finishing action at xTier™ termination

Note: As a result of the fact that the JAVA and .NET xTier™ implementations have a number of distinctions, the execution environment context for each of these runtimes will contain slightly different elements. The following examples are specific to the JAVA xTier™ implementation.

xTier™ Kernel Start

The xTier™ kernel should be started by the micro kernel because, depending on the particular hosting environment, the kernel start can be performed in various ways. For example, in standalone mode the system kernel start is performed simply in a separate thread, but while working with application servers the xTier™ kernel is started as a service of the application server.



Passing the Environment Context

By passing the execution environment context to the xTier™ Kernel when starting, the xTier™ Kernel services can take environment features into account when initializing. When implementing the environment context, the developer must implement the following interface:

```
public interface MicroKernelContext {
    /** Hosting environment: J2EE app server. */
    public static final int HOST_ENV_J2EE = 1;

    /** Hosting environment: JSP/Servlet engine. */
    public static final int HOST_ENV_JSP = 2;

    /** Hosting environment: standalone xTier runtime. */
    public static final int HOST_ENV_XTIER = 3;

    /** Hosting environment: unknown or undefined. */
    public static final int HOST_ENV_UNKNOWN = 4;

    ....
}
```

```
public int getHostEnvId();

public Locale getLocale();

public MBeanServer getMBeanServer();

public MicroKernelLogger getLogger();

public String getXtierRoot();

public String getKernelRegion();

public void onKernelShutdown() throws MicroKernelException;
}
```

There are a number of mandatory elements:

- **Execution Environment:**
 - J2EE application servers
 - JSP/Servlet containers
 - In-process (stand alone)
- **Locale:** This sets the default for the Java VM. The Java default system locale is used in the absence of setting this.
- **MBean Server:** This is used for JMX services management. In the pre-built micro kernel for use with an application server, the MBean server for the given application server is returned, for the standalone mode microkernel an MBean server is created directly in the microkernel and registered using a default HTML Adaptor. (See **com.sun.jdmk.com.HtmlAdaptorServer**.)
- **Logger:** Please see below for additional details.
- Starting Parameters:
 - **XTIER_ROOT** – the file system folder relative to the xTier configuration files are located.
 - **xTier Kernel Region Name** – The kernel region name corresponds to the kernel region defined in the **xtier_kernel.xml** configuration file and determines the appropriate configuration for the xTier™ Kernel defined in the configuration files.

Additionally, the **onKernelShutdown()** method should be defined. This method will perform any final actions for the given environment when the xTier™ kernel stops. Note: The exception **MicroKernelException** thrown by the **onKernelShutdown()** method is part of the xTier™ Micro kernel API and allows processing exceptional situations that might arise in the micro kernel.

Stopping

The micro kernel implementation should define the following necessary tasks:

- stop the xTier™ Kernel
- perform any necessary actions at end of system work, depending on environment. This task should be resolved at execution environment context definition (See “Passing the Environment Context above.”)

Logging in xTier™

Notice in the preceding section that the **MicroKernelContext** specifies a logging implementation. When developing a micro kernel the developer should implement the **MicroKernelLogger** interface. The logger implementation in the execution environment has special importance as the micro kernel does the bulk of its work during system start and stop when it is necessary to know about the success and failure of these operations. The **MicroKernelLogger** interface follows:

Interface MicroKernelLogger:

```
public interface MicroKernelLogger {  
    public void log(Object msg);  
  
    public void log(Object msg, Throwable e);  
  
    public void error(Object msg);  
  
    public void error(Object msg, Throwable e);  
  
    public void trace(Object msg);  
  
    public void trace(Object msg, Throwable e);  
}
```

```
public void info(Object msg);  
  
public void info(Object msg, Throwable e);  
  
public void warning(Object msg);  
  
public void warning(Object msg, Throwable e);  
  
public void debug(Object msg);  
  
public void debug(Object msg, Throwable e);  
  
public MicroKernelLogger getLogger(String ctgr);  
}
```

The interface defines six different levels of log messages, corresponding to the xTier™ Log Service message types: debug, error, log, trace, warning and info. The interface also defines a method which returns the logger for an instance of one of these categories.

When implementing the **MicroKernelLogger** interface the developer should redirect log messages to the execution environment's logging system. Most application servers have their own logging system. For example, the JBoss micro kernel redirects log messages to the JBoss logging system. On the other hand, in the case of a deployment environment that does not have an inherent logging mechanism the developer is free to implement the logging as they see fit (standard out, files, RDBMS, email, etc.). In the J2SE in-process micro kernel implementation, for example, the log messages are output to the standard streams **System.out** and **System.err**. It is possible, although not recommended that certain levels of the messages can be ignored by the micro kernel implementation.

The following is a simple example of a logger implementation:

```
public class UserMicroKernelLogger implements MicroKernelLogger {
    ....
    // Ignore it.
    public void debug(Object msg) {
        // No-op.
    }
    ....
    public void error(Object msg) {
        System.err.println(msg);
    }
    ....
}
```

Sample Micro Kernel – Starting the xTier™ Kernel

The following example illustrates the process of starting the xTier™ kernel from within a Java microkernel.

```
public class UserMicroKernel {
    ....
    /**
     * Starts xTier kernel with this micro kernel initialized with given micro kernel's
     * parameters.
     */
    public static void start(final StandardMicroKernelParams params) throws
    MicroKernelException {
        ....
        final XTierKernel xtier = XTierKernel.getInstance();
        ....

        Thread startup = new Thread("kernel-startup") {
            /**
             *
             */
            public void run() {
                ....

                MicroKernelContext microKernelContext = new MicroKernelContext() {
                    ....
                }
            }
        }
    }
}
```

```
Map props = new HashMap(1);
props.put(MicroKernelContext.MICRO_KERNEL_CONTEXT,
microKernelContext);

try {
    /**
     * Start the xTier kernel.
     * -----
     */
    xtier.start(props);
}
// Guard against any error condition during kernel startup.
catch (Throwable e) {
    ....
}
....
}
```

Note: In this example, the developer should hold a reference to the kernel instance (or obtain this reference again) in the program and call the kernel **stop()** method at the end of application execution. This follows:

```
....
// Stop the xTier™ kernel.
xtier.stop();
....
```

Please note that the examples given show only the basic schematic implementation of a micro kernel and consider only the base interaction mechanism of the micro kernel, kernel and execution environments. For more detailed code examples, with features such as the creation of the execution environment context, log-redirection etc., please see the micro kernel code examples contained in the xTier™ distribution.

Sample Micro Kernel – Jboss 3.x

The following example contains the interface and class implementing a micro kernel for the JBoss 3 application server. (Note: To get familiar with the JBoss API, please see the appropriate product documentation located at: <http://www.jboss.org>)

```
public interface JBoss3MicroKernelMBean extends
org.jboss.system.ServiceMBean {
    ....
    public String getKernelRegion();

    ....
    public void setKernelRegion(String kernelRegion);

    ....
}

public class JBoss3MicroKernel extends org.jboss.system.ServiceMBeanSupport
implements JBoss3MicroKernelMBean {
    ....
    /**
     * @see org.jboss.system.ServiceMBeanSupport#startService()
     */
    protected void startService() throws Exception {
        ....

        // Getting kernel's reference.
        XtierKernel xtier = XtierKernel.getInstance();
        ....

        MicroKernelContext microKernelContext = new MicroKernelContext() {
            ....
        };

        Map props = new HashMap(1);
```

```
        props.put(MicroKernelContext.MICRO_KERNEL_CONTEXT,
microKernelContext);

        try {
            /*
             * Start the xTier kernel.
             */
            xtier.start(props);
        }
        catch (KernelException e) {
            throw new Exception("xTier failed to start.", e);
        }
        ....
    }

    /**
     * @see org.jboss.system.ServiceMBeanSupport#stopService()
     */
    protected void stopService() throws Exception {
        ....
        // Getting kernel's reference.
        XtierKernel xtier = XtierKernel.getInstance();

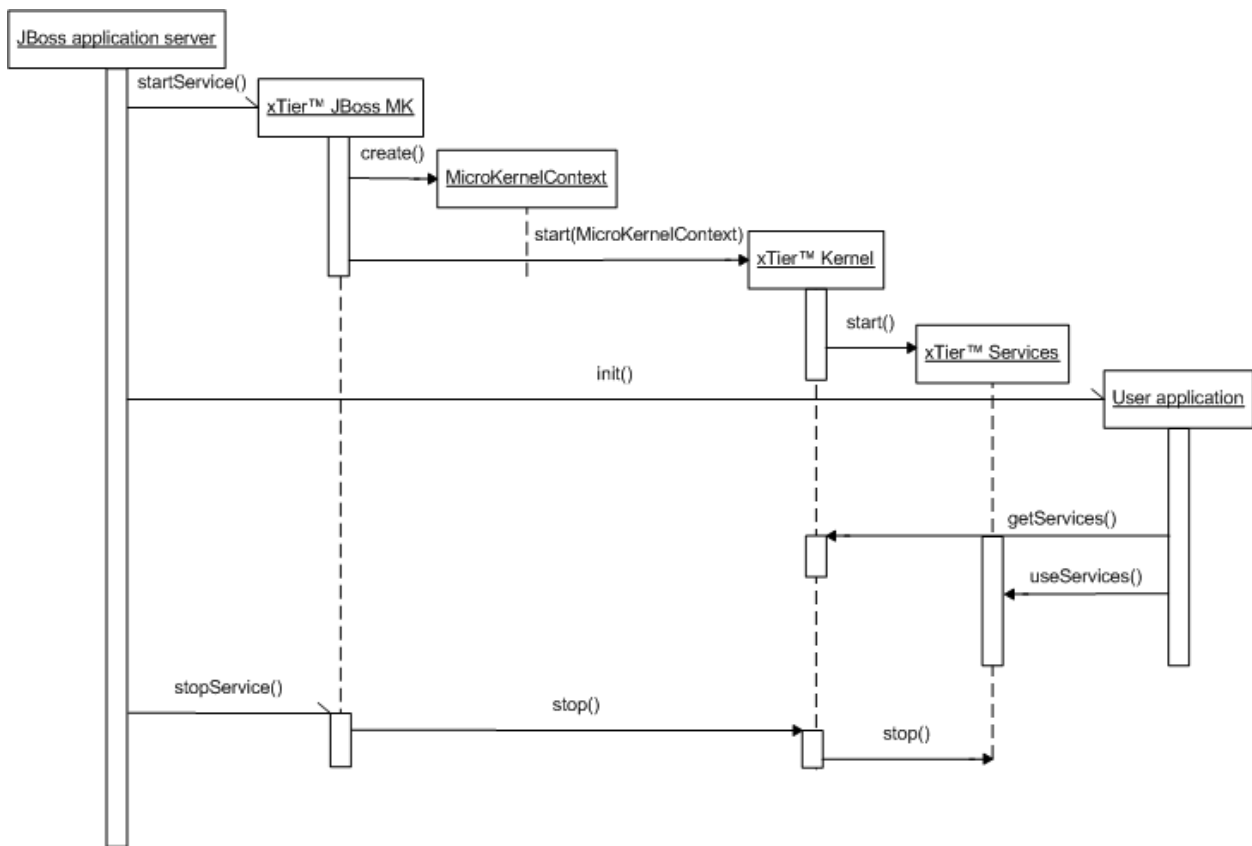
        if (xtier.getState() == XtierKernel.KERNEL_STARTED) {
            xtier.stop();
        }
    }
}
```

Note: in order to simplify the example, only the methods associated with setting and getting a kernel region are shown. (Please see the preceding section for a description of the environment context.)

- The interface (**JBoss3MicroKernelMBean**) which is used for JMX management of the service is implemented using inheritance from **org.jboss.system.ServiceMBean**.
- The class (**JBoss3MicroKernel**) implementing this interface and expanding **org.jboss.system.ServiceMBeanSupport** is implemented.

- This class is responsible for starting and stopping the xTier™ kernel.
- The method **startService()** overrides the appropriate method from **org.jboss.system.ServiceMBeanSupport** and is responsible for starting the xTier™ kernel, and correspondingly the method **stopService()** overrides the respective method from the ServiceMBeanSupport class and is responsible for stopping the xTier™ Kernel.

The following diagram illustrates the life cycle of an application running on JBoss.



The description of the **JBoss3MicroKernel** class should be placed in the standard JBoss deployment descriptor. This descriptor and the set of appropriate jar's file gets placed in the JBoss deployment directory. The methods **startService()** and **stopService()** carry out all necessary functions to start and stop xTier™ and are executed using the standard JBoss mechanisms.

An example of a deployment descriptor is given below:

```
<server>
<classpath codebase="."
  archives="
    xtier/xtier.jar,
```

```
xtier/xtier-jboss3-mk.jar,  
... "/>  
  
<mbean code="JBoss3MicroKernel" name="xtier:service=xtier-jboss3-mk">  
.... Set of attributes  
</mbean>  
</server>
```

Sample Micro Kernel – WebLogic 8 Application Server

An example of the classes implementing a micro kernel for the WebLogic 8 application server is given below. (Note: To get familiar with the WebLogic API read the appropriate product documentation located at <http://www.bea.com>)

```
public class Weblogic8MicroKernelStartup implements  
weblogic.common.T3StartupDef {  
....  
  
/**  
 * @see weblogic.common.T3StartupDef#startup(String, Hashtable)  
 */  
public String startup(String name, Hashtable args) throws Exception {  
....  
// Getting kernel's reference.  
XtierKernel xtier = XtierKernel.getInstance();  
  
MicroKernelContext microKernelContext = new MicroKernelContext() {  
....  
};  
  
Map props = new HashMap();  
  
props.put(MicroKernelContext.MICRO_KERNEL_CONTEXT,  
microKernelContext);  
  
try {  
/**  
 * Start the xTier kernel.  
 */  
xtier.start(props);  
}  
catch (Throwable e) {  
throw new Exception("xTier failed to start.", e);  
}
```

```
}  
  
return "xTier WLS 8 micro kernel started ok.";  
}  
}  
  
public class Weblogic8MicroKernelShutdown implements  
weblogic.common.T3ShutdownDef {  
....  
  
/**  
 * @see T3ShutdownDef#shutdown(String, Hashtable)  
 */  
public String shutdown(String name, Hashtable args) throws Exception {  
// Getting kernel's reference.  
XtierKernel xtier = XtierKernel.getInstance();  
  
if (xtier.getState() == XtierKernel.KERNEL_STARTED) {  
xtier.stop();  
  
return "xTier WLS 8 micro kernel stopped ok.";  
}  
else {  
return "xTier kernel is not started.";  
}  
}  
}
```

- The **weblogic.common** library contains, among other things, 2 interfaces: **T3StartupDef** and **T3ShutdownDef**. By implementing them it is possible to create applications that are executed at the startup and stop of the WebLogic server respectively.
- In this example, these 2 classes are implemented:
 - **Weblogic8MicroKernelStartup** and **Weblogic8MicroKernelShutdown** - These classes should be loaded in the WebLogic application server and paths to them should be registered, using the Weblogic Server Console, in the "Deployments/Startup and Shutdown" section. Once these classes are registered,

the **Weblogic8MicroKernelStartup.startup (...)** and **Weblogic8MicroKernelShutdown.shutdown (...)** methods, carrying out all the necessary functions to start the xTier™ Kernel and are executed using WebLogic's own mechanisms.

Note: Using the Weblogic Server Console is not the only way to specify classes to be executed during system start and at shutdown. It is enough to register the StartupClass and ShutdownClass attributes in the appropriate Weblogic domain in the config.xml configuration file.

Example configuration:

```
<StartupClass
  Arguments="Agruments list"
  ClassName="Weblogic8MicroKernelStartup"
  FailureIsFatal="false" LoadBeforeAppDeployments="true"
  Name="startup" Targets="myserver"/>

<ShutdownClass
  ClassName="com.fitechlabs.xnova.wls8.Weblogic8MicroKernelShutdown"
  Name="shutdown" Targets="myserver"/>
```

The life cycle of applications running under WebLogic is similar to the life cycle of those running under JBoss except that under WebLogic the microkernel has no constant lifetime, because unlike JBoss it does not register itself as a service.

Conclusions

This whitepaper has provided examples of several different xTier™ micro kernel implementations and has shown how by using a micro kernel that integration with various application servers and environments that have completely different principles of deployment and execution are greatly simplified. As a result of choosing a micro kernel design, (i.e. the xTier™ subsystem that deals with environment specific interaction), in all of these cases the xTier™ Kernel usage remains transparent, while at the same time, the micro kernel implementation is not overly labor-intensive. This approach to integration makes the use of micro kernel technology for binding xTier™ with a wide variety of environment as the most convenient way to increase developer flexibility. Further, since the way in which the xTier™ Kernel services are access and used remains constant, the developer is not forced to remain tied to a given platform, as the underlying platform can be easily changed by replacing the only component of the system that is unique to that environment – the micro kernel, while the remaining code base does not change. This approach gives the developer the broadest opportunity for development, testing and deployment of their applications.

Fitech Laboratories Inc.

Corporate Headquarters

300 Montgomery St., Suite 621
San Francisco, CA 94104
USA

Phone: 1-415-371-8234
Fax: 1-415-371-8237

East Coast Sales Office

330 Madison Ave., 9th floor
New York, NY 10017
USA

Phone: 1-646-495-5076

Fitech Laboratories Japan

Toranomon40 MT Bldg. 3F
5-13-1 Toranomon Minato-ku
Tokyo 105-0001, Japan

Phone: +81-3-5402-7711
Web: www.fitechlabs.co.jp