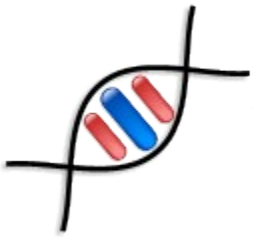


Xito BootStrap Developers Guide

prepared by: Deane Richan

Covers Version: 1.1.5

Table of Contents

Introduction

Since the introduction of Java technology, developers have used Java to create client applications. In the beginning Java Applets embedded in web pages were used to deploy application functionality. Later many Java developers began to write fat client applications that are installed and run directly from the local computer. More recently the Java Network Launching Protocol (JNLP) has been introduced to allow full applications to be deployed through a link on a web page.

Although each of these approaches have their place there continues to be a need for better client platform Java integration.

Xito BootStrap addresses these issues by providing the following basic functionality:

- Provide a Simple mechanism to launch applications and services from remote Internet Servers
- Provide an elegant and seamless security model enabling network deployed applications to be executed in a secure environment
- Provide functionality to deploy Shared Services and applications running in a single VM to reduce memory requirements and improve startup times.

These three high level goals can be described simply as: **Remote Launching, Secure Execution, and Shared VM.** In addition, an important aspect of the approach followed by the BootStrap is seamless native integration, both in terms of user interface and functionality.

This document describes how developers can use the BootStrap to deploy their applications and create execution environments that take advantage of BootStrap's built-in abilities.

BootStrap Architecture

The Xito BootStrap is the entry point to the larger Xito platform. Although BootStrap is designed to boot the Xito platform it can be used to boot any application or set of application services.

The BootStrap initializes the environment, sets up the security manager, and then loads a set of services. Services are just Java Applications and don't require any specific interface implementation.

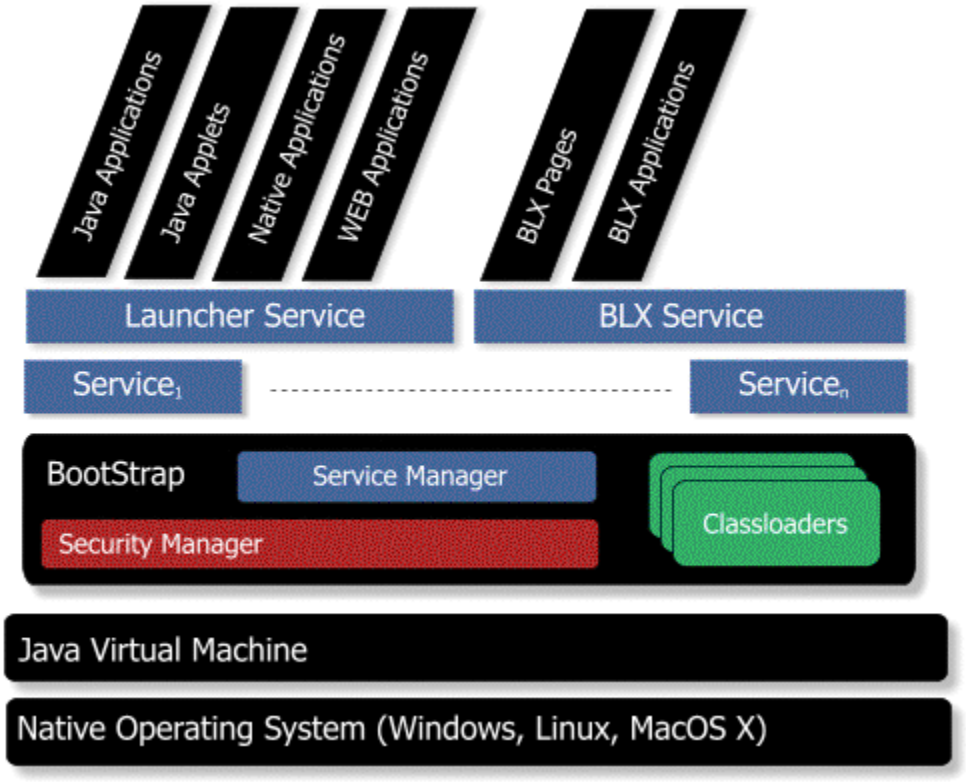
The BootStrap launches services and applications into a single Java virtual machine. This reduces memory requirements of all applications and services running in the environment, and also enables very quick execution of downloaded applications.

Sharing of a single Virtual Machine increases performance and reduces memory requirements and allows untrusted code to run along side trusted code. However sharing a VM requires applications to cooperate with relation to the VM resources. Applications that want full control over the virtual machine can be launched into a new VM instance from the bootstrap.

Applications and services running in the BootStrap are designed to be executed directly from a web server. Therefore there is no installation required for services or applications. They are also always up to date because they are running from remote servers.

A caching mechanism is used in the BootStrap to cache Java applications and service resources on the local hard drive so that they do not need to be downloaded each time they are executed. Also the local cache enables applications to be executed when the client is not connected to the network, a feature not available for traditional HTML based web applications. This network launching functionality allows applications deployed through Xito to have the benefits of a web application and also the benefits of a rich client application.

The following diagram describes the major components of the Xito Platform.



The Xito BootStrap is the entry point to the platform. The BootStrap is a Java Application that initializes the environment, reads system properties, starts a Security Manager, and then executes a set of services.

The BootStrap also handles all loading of application resources and execution of applications. It provides this functionality through four main components: Service Manger, Cache Manager, ClassLoaders and Executable Descriptors. A separate Launching Service is provided to perform more complex application launching such as Java Applets or JNLP based Java applications.

Bootstrap also contains a win32 native executable that can be used to launch the application on Windows platforms. The native exe can be renamed to what ever application name the developer wants for their application.

BootStrap Installation

System Requirements
JRE 1.4.2+
Windows, MacOSX, Linux x86

Note: BootStrap has been primarily written and tested on the Windows Platform. MacOSX and Linux should also work but testing efforts have not been completed on those platforms.

License
Xito BootStrap is distributed under the Apache 2.0 License See:
http://www.apache.org/licenses/LICENSE-2.0 for more information.

Obtaining the BootStrap

Visit the **http://xito.sourceforge.net/projects/bootstrap** and choose **download**.
Download the most recent version of BootStrap. This document is based on version 0.9.2.

Unzip the BootStrap into a directory of your choosing. The following files are included:

Filename	***Description***
boot.jar	Main Jar of the BootStrap environment
boot.properties	BootStrap properties to initialize this environment
boot_services.xml	List of Boot Services
sample.jar	Jar of Sample Service

Filename	*Description*
sample.srv	Sample Service Descriptor
start_services.xml	List of Startup Services
bootstrap.exe	Win32 Launcher
bootstrap_console.exe	Win32 Launcher with console output
license.html	CDDL License Info

Note: The bootstrap.exe and bootstrap_console.exe are provided with the Win32 bootstrap release. These executables will attempt to locate the Java VM using the Windows Registry and then boot the environment. They can be renamed to your application name if you prefer.

To test the installation of the BootStrap run one of the native EXEs or at a command prompt in the bootstrap directory execute: **java -jar boot.jar**

You will first be prompted to configure your Proxy Server. This will just occur the first time you execute this application configuration.

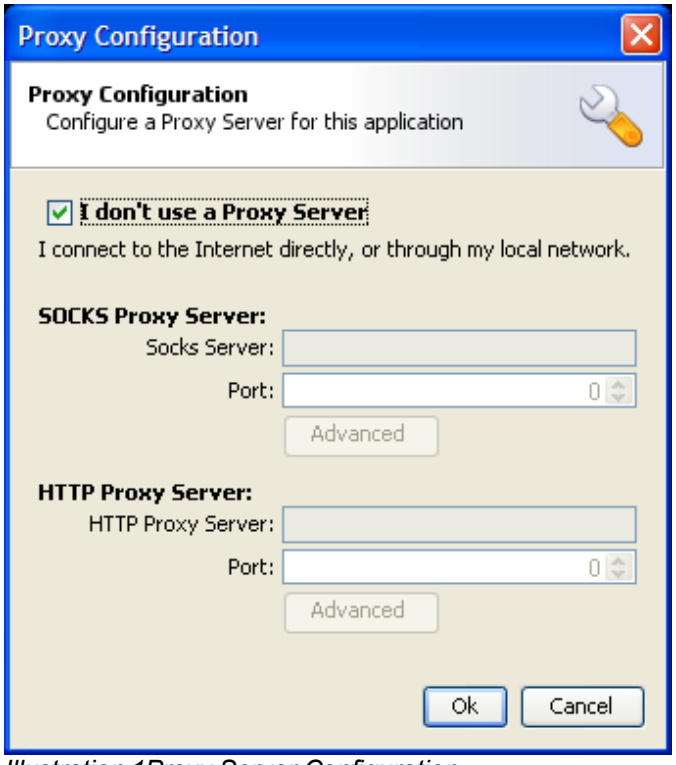


Illustration 1Proxy Server Configuration

After you configure the Proxy Settings for your machine the Sample Service should execute. This sample Service is a simple application launcher that will launch the Java SwingSet from the Sun web servers or launch the Asteroids games from the Xito web server.

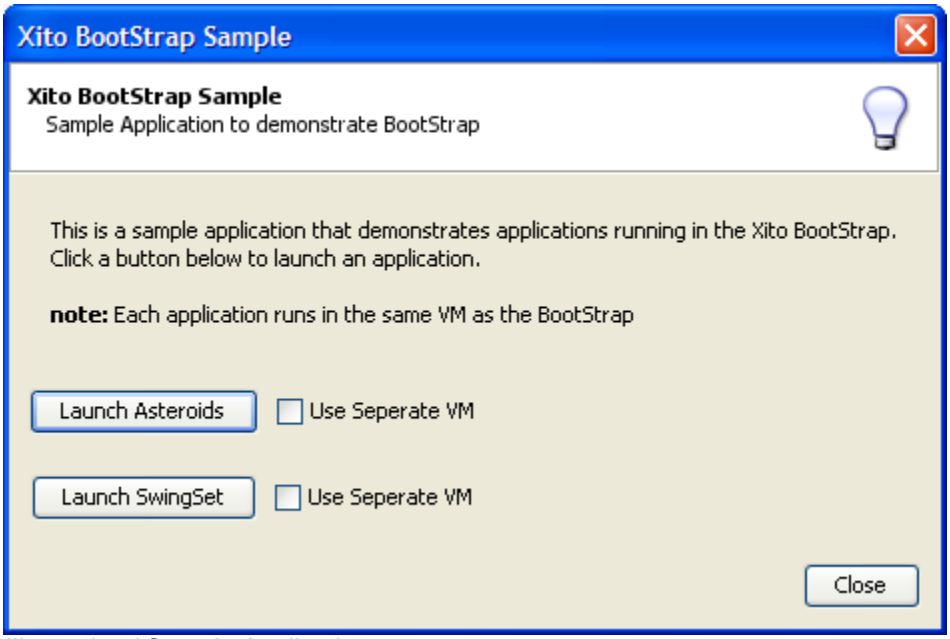


Illustration 2Sample Application

When executing any code under Bootstrap that does not exist in the boot directory, the user will be prompted to grant permissions to that application or service.

With the sample application if the user chooses to launch asteroids or SwingSet they will be prompted to grant permissions to those applications.

Note: SwingSet will not execute correctly if it is not granted All Permissions. The Asteroids sample application will execute if it is denied permissions but will be running in a restricted environment. The Asteroids window will contain a restricted content warning banner and its full screen mode will be disabled.

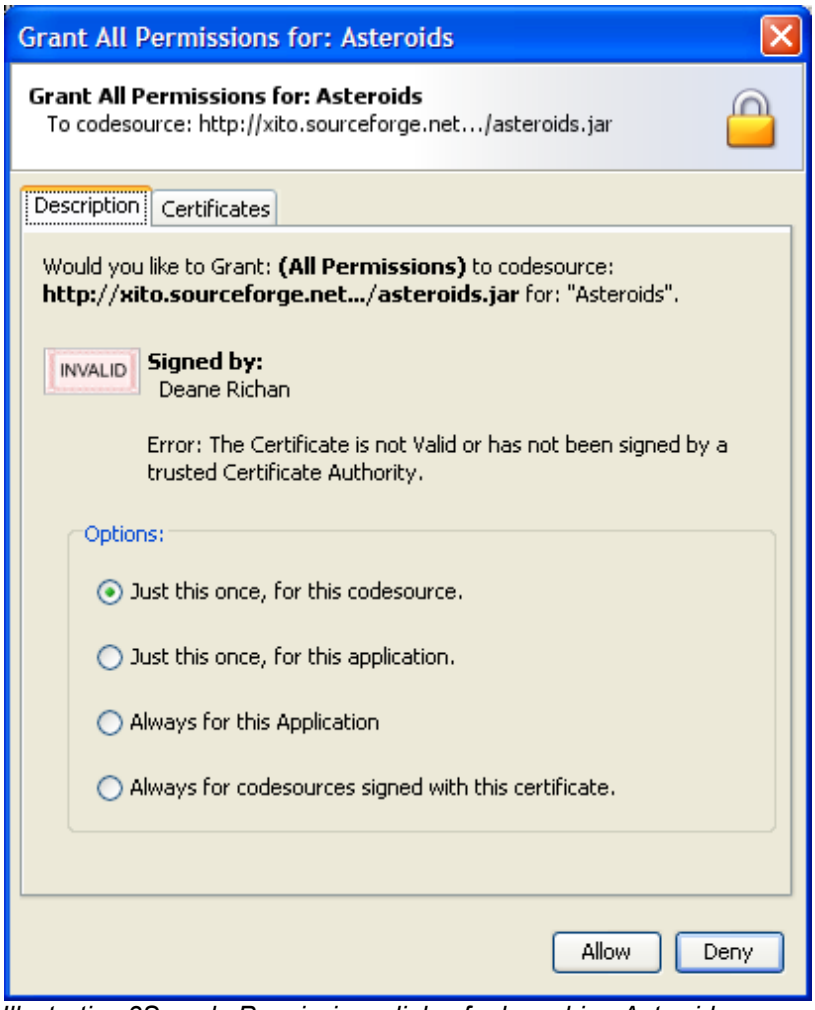


Illustration 3Sample Permissions dialog for launching Asteroids

Using this dialog the user can view the code source of the application and whether it was signed and by whom. They can then Allow the permission or Deny it.

Boot Sequence of a BootStrap Application

To launch a BootStrap configured application you simply execute the BootStrap using:

java -jar boot.jar

> *Note: On Windows a native EXE launcher is provided that automatically determines the location of the installed JRE and launches the environment. When using this native launcher the bootdir defaults to the location of the EXE.*

This will start the boot process using the following sequence:

Determine Boot Dir

The **bootdir** is the directory where the bootstrap configuration is located. By default the **bootdir** is the current directory. You can specify a different **bootdir** by passing:

-bootdir [boot directory]

example:

java -jar boot.jar -bootdir c:/temp/test_boot_dir

If the bootdir is not a valid directory then the boot process will fail with an error.

Read boot.properties

After the bootdir is located a boot.properties file is read from that directory. A boot.properties file must exist. If it does not the boot process will fail with an error.

Information about the settings in the boot.properties are explained in the **Configuring Bootstrap** section below. Note: All settings in the boot.properties file are also loaded into the System.properties object. Therefore you can specify any Java System property in the boot.properties file.

Read logging.properties

If a logging.properties file is found in the bootdir it will be processed to configure the logging settings of the environment. BootStrap uses the built in java.util.logging package for log messages. For information on configuring the logging.properties file see the Java Logging API documentation in the class: java.util.logging.LogManager

Setup Security Manager

The Security Manager is then initialized. BootStrap uses built in SecurityManager of the class: org.xito.boot.BootSecurityManager. More information about BootStrap security is found in the **Security Configuration** section.

Setup the Cache Manager

The cache manager is initialized to create a local directory where all downloaded resources will be cached. This directory is configurable but by default is relative to the users home directory.

Proxy Configuration

The first time the application is executed the user will be prompted for a Proxy Configuration. This prompt is skipped if proxy settings are already contained in the boot.properties

Service Startup

The services are then started using the boot_services.xml and start_services.xml files located in the bootdir. These files list the service applications or a single application that should be started by the bootstrap.
All Boot Services must execute without exception for the environment to boot. All Startup Services are considered optional and therefore don't need to execute successfully for the boot process to succeed.

UI Check

Because BootStrap is a client execution framework the assumption is that a GUI will be presented to the user by one of the running services. When BootStrap has finished launching all services it will attempt to determine if a UI Window has been created. It does this by creating a timer to check for a UI Window after 20 seconds of the Services being started. If a UI window can not be found then the Boot process will fail with an Error. This behavior can be turned off by running BootStrap in the **nogui mode**.

After the services are started the environment has been booted and should be up and running.

Configuring BootStrap

The main Bootstrap configuration consists of three separate configuration files. These include: **boot.properties, boot_services.xml** and **start_services.xml**.

These files specify how the environment is to initialize and what services should be started. Also a limited number of Startup Arguments can be passed on the command line.

Startup Arguments

The following startup arguments can be passed to the bootstrap command line.

Argument	***Description***
-bootdir	Directory where the boot files will reside
-nogui	Defaults to false. If true then Bootstrap will not display any UI Prompts or messages. See **boot.nogui** boot property for more information
-minmode	Start the environment in minimum mode. This will cause only boot_services to be started and only those flagged as: **<minimum-srv>true</minimum-srv>** in the boot_services.xml file

Configuring boot.properties

The boot.properties settings contains options to configure the BootStrap environment. The file can also contain any Java System property because all settings in the boot.properties file will be loaded into System.properties. The file should be located in the boot dir of the application.

The following boot.properties settings can be specified in the file

Property	***Description***
app.name	**(required)** The app.name property specifies the name of this application. This name should be a short variable style name. For example if your application is called “ACME SQL Editor” then the app.name property should be something like: acme_sql_editor
app.display.name	**(Optional)** The app.display.name is used to display the name of your application in prompts or other UI elements. This defaults to the value of app.name if not specified. Using the example from app.name you could specify the app.display.name to be “ACME SQL Editor”
app.icon	**(Optional)** The default Icon that is used for all UI windows in the environment. This icon will replace the normal Java Icon unless you explicitly specify your window icons in your java code. This setting is a URL location that is considered to be relative to the bootdir.
app.base.dir	**(Optional)** The directory where this application stores all of its settings: Preferences, Security Permissions, Cache etc. By default this directory will be {user.home}/.{app.name} *Example: On windows for a user named Deane with an application named my_app the app.base.dir would be:* c:/Documents and Settings/Deane/.my_app
boot.nogui	**(Optional)** Defaults to false. Same as the nogui startup argument. Setting this to true will cause all of the BootStrap UI to be disabled. Any Security Dialog or Error message that would have been shown to the end user will simply be logged instead. Also if this option is set the checkUI procedure in the boot process will be skipped. Note: This option doesn't stop services or launched applications from having a UI. To disable all UI use java specific: **java.awt.headless** option.

Property	***Description***
boot.no.security	**(Optional)** Defaults to false. If set to true the BootStrap SecurityManager will not be installed. This is not recommended. With no security manager all downloaded code will have all permissions, and will be able to access the local system. Create Windows without warning banners, and be able to exit the VM even if other apps are running
boot.use.cache	**(Optional)** Defaults to true. Setting this to false will cause the Cache Manager to not store downloaded resources in the local cache.
boot.cache.dir	**(Optional)** This is the directory where the cache manager will stored downloaded resources. This directory will default to {app.base.dir}/cache
boot.proxy.prompt	**(Optional)** Defaults to true. If this setting is set to false then the user will not be prompted for Proxy settings on the first launch.
native.laf	**(Optional)** Defaults to true. By default bootstrap will use the native platforms look and feel for its GUI. Setting this to false will cause Java to use its default look and feel which is usually the Swing Metal Look and Feel. *Note: Even if BootStrap is using the native look your launched applications can use their own look and feel because of the AppContext option.*
boot.services.url	**(Optional)** This is the URL to the boot_services.xml file that this BootStrap should boot. If not specified it will default to boot_services.xml located in the bootdir.
startup.services.url	**(Optional)** This is the URL to the start_services.xml file that this BootStrap should startup. If not specified it will default to start_services.xml located in the bootdir.
quicklaunch.install.dir	**(Optional)** When launching BootStrap from WebStart this is the directory where the quicklaunch files will be copied to on the client. See the Launching from WebStart section for more information.

As the table above describes the only setting that is required is the **app.name**. All other settings are optional.

Launching BootStrap from Java WebStart

The Xito BootStrap is designed to work in place of the Java WebStart therefore launching a BootStrap based application from WebStart is error prone. However at times it may be useful to have WebStart launch your application. Therefore starting with the 1.0.1 release, the ability to launch a BootStrap based application environment from WebStart is now supported.

The functionality in the BootStrap to enable this is called QuickLaunch. Basically you can provide a JNLP file that will launch the BootStrap. When BootStrap sees that it has been launched by a JNLP Client such as WebStart it will tricker the BootStrap to go into QuickLaunch Mode.

While in QuickLaunch Mode the BootStrap will obtain a list of Boot Files to install on the client from the server and will cache and copy these files to the local machine. It will then launch a new instance of the BootStrap based on these copied files.

Steps for QuickLaunch deployment:

1. Deploy BootStrap and Service files on WebServer
2. Modify QuickLaunch JNLP File
3. Create QuickLaunch File List
4. Modify the boot.properties

Deploy BootStrap Service files on Web Server

Follow normal BootStrap deployment information to create your services and deploy them on your web server for your clients to use.

QuickLaunch JNLP File

To use QuickLaunch you need a JNLP file that webstart can use to launch the BootStrap. The BootStrap distribution contains a sample JNLP file for this purpose called quick_launch.jnlp.

Sample quick_launch.jnlp file:

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- NOTE: The codebase should be changed to your codebase for deployment -->
<jnlp spec="1.0+" codebase="http://quick_launch.test/">
    <information>
        <title>Xito</title>
        <vendor>Xito.org</vendor>
        <description>Xito Application Platform</description>
        <description kind="tooltip">Xito Application Platform</description>
        <homepage href="http://xito.sourceforge.net"/>
        <offline-allowed/>
    </information>
    <security>
        <all-permissions/>
    </security>
    <resources>
        <j2se version="1.4+"/>
        <jar href="boot.jar" main="true"/>
        <property name="quicklaunch.app.display.name" value="Xito"/>
    </resources>
    <application-desc main-class="org.xito.boot.Boot">
    </application-desc>
</jnlp>
```

The code base should be the location on the webserver you have placed your boot.jar and boot.properties files.

You can also modify the <property name=”quicklaunch.app.display.name”> to change the name of the application that the BootStrap will use while its in QuickLaunch mode.

Create a QuickLaunch File List

In the codebase directory on the webserver you should place a file named: quick_launch_file_list.txt, this file will list the files that need to be copied from the WebServer to the client in order for BootStrap to be launched. You can list just the BootStrap if you want all services to launch from the web server or you can have individual service files also listed. A sample quick_launch_file_list.txt file is shown below:

Sample quick_file_list.txt:

```
#List each file that should be copied to the local machine
#These filenames should be relative to the JNLP codebase
#specified in quick_launch.jnlp

boot.jar
boot.properties
xito_16.png
```

The files boot.jar and boot.properties are required. These are the minimum files that you need to start the BootStrap. The files listed should be placed on the webserver relative to the codebase specified in the JNLP file. Lines starting with # are considered comments in this file.

Modify the boot.properties

Using the QuickLaunch File List the BootStrap will copy the files listed to the local machine however it needs to know which directory on the local machine to copy the files to. To specify this directory you need to add a property to the boot.properties file called: quicklaunch.install.dir

This property should name a directory on the client where the files listed in the File List will be copied. The directory name should not be a full path just a name for your application.

Sample boot.properties with quicklaunch.install.dir propety:

```
app.name = xito_test
app.display.name = Xito Test App
app.icon = xito_16.png

boot.services.url=http://xito.test/test_app/boot_services.xml
startup.services.url=http://xito.test/test_app/start_services.xml

quicklaunch.install.dir=Xito
```

On the Windows platform the quicklaunch.install.dir will be placed under "c:\Program Files" on MacOSX the directory will be placed under "/Applications".

Once the files have been copied to the install directory BootStrap will launch the boot.jar that it copied to the install directory using a new Java VM process. The VM will be the same version that was used by the WebStart application.

From this point execution resembles the normal BootStrap boot process.

Boot and Startup Services

After the Boot process has configured the environment, and setup the cache manager and security manager, it will boot a set of specified Services. Services are simply Java applications that have been configured to be started by the BootStrap. There are two sets of Services: boot services and startup services. Boot Services are required to execute with out exception for the environment to boot successfully. Startup Services are started after Boot Services and do not have to start successfully for the environment to complete the boot process. In other words, Boot Services are required services and Startup Services are optional services.

Note: If the environment is in minmode only boot services specified as minimum services will be started. Startup Services will be skipped.

Services XML File

The Services that are to be started are listed in two files boot_services.xml and start_services.xml. By default these files should reside in the bootdir however remote files can be used by specifying **boot.services.url** and **startup.services.url** in the boot.properties file.

The content of each of these files lists the services that should be started. They will be started in the order they appear in the XML file. An example Services XML File is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<services>
   <service>
      <name>test.sample.MyApp</name>
      <display-name>Sample App</display-name>
      <minimum-srv>false</minimum-srv >
      <href>sample.srv</href>
   </service>
   <service>
    ...
   </service>
</services>
```

There are four settings that can be specified for each service, two are required. They are

Element	*Description*
name	**(Required)** This is a name of the service. The name should follow Java's package naming structure. The name will uniquely identify the service to the Service Manager and be use to resolve dependent services class dependences.
display-name	**(Optional)** A Descriptive Name of This service
minimum-srv	**(Optional)** If set to true then this service will be started if BootStrap is in minmode. Min mode can be configured by passing a -minmode command line argument. The default is false.
href	**(Required)** A URL pointing to a Service Descriptor .srv file for this Service. The URL can be absolute or relative to this services.xml file

The Service Manager will iterate through each **<service>** element and load each service in turn. The **<href>** element should point to a valid Service Descriptor XML file for each Service. By convention each individual Service Descriptor file should have a .srv extension.

Service XML (srv) File

The Service Descriptor file describes each Service in detail including the jars that the service will use to load and also the name of the main service class. It can also contain a list of services that this service depends on, using the service-ref element in the classpath element. If the service in service-ref has not been loaded it can be automatically loaded by using the href attribute of the service-ref element.

An example Service Descriptor XML File is shown below:

```
<?xml version="1.0" encoding="UTF-8"?>
<service>
   <name>test.sample.MyApp</name>
   <display-name>Sample Application</display-name>
   <desc>This is a Sample Application</desc>
   <version>1.0.0</version>
```

```
   <service-cls>test.sample.MyApp</service-cls>
   <append-to-classpath>true</append-to-classpath>
   <classpath>
      <service-ref name="test.shared.service"/>
      <lib path="sample.jar"/>
   </classpath>
</service>
```

The Elements in the Service Descriptor file are described below:

Element	***Description***
name	**(Required)** The name of the service this should be the same name that was listed in the file for **boot_services.xml** or **start_services.xml**
display-name	**(Optional)** A Descriptive Name of this Service
desc	**(Optional)** A Description of what this Service is or what it does
version	**(Optional)** xx.xx.xx version number of this service
service-cls	**(Optional)** This is the class name that contains the **main** method of the service
append-to-classpath	**(Optional)** If this option is set to true then this services classes will be included in the classpath for all following services or applications launched in this VM. By using this option Applications and Services don't need to explicitly declare their service dependancies
classpath	**(Optional)** A list of jar libraries or services that should be included in this Services classpath. This element uses two sub-elements named: **lib** and **service-ref**
Native-libs	**(Optional)** A list of jar libraries that contain native operating system libraries. This element will contain a lib sub-element.

Element	*Description*
lib	**(Optional)**This element must reside under a classpath element and names a jar library that should be included in the classpath. The attribute **path** is used to name the path to the jar file. The path can be absolute or relative to this Service Descriptor file. If the element is used under a native-libs element the attribute: os can be used with the value “windows”, “mac”, or “linux”
service-ref	**(Optional)** This element must reside under a classpath element and names a service that should be included in this services classpath. The attribute **name** is used to name a service that should be included. The service that is named must have been already started by the service manager or an **href** attribute can be used to have the service loaded automatically if it hasn't yet been loaded

Each service is configured to require All Permissions. Therefore when services are launched from locations other then the bootdir the user will be prompted to grant the service permissions.

To create a service the developer simply needs to write a standard Java Application with a **main** method. Bundle that application into a jar and then create a Service Descriptor file for it. Then list the service in either the boot or startup list of services. No interfaces or BootStrap APIs need to be used.

Using BootStrap API in your Service

The service can take advantage of the BootStrap API if it so chooses. There are several ways that a service can more tightly integrate with the BootStrap:

- Implement a **static initService** method in your main Service class
- Register a **StartupListener**
- Use Launcher API's to launch separate embedded applications

To access this functionality in your Service you can **import org.xito.boot.***

initService Method

If the main Service class has implemented an initService method of the signature:

public static void initService(ServiceDesc service)

Then the ServiceManager will call this method on your service just prior to calling the static **main** method. The passed ServiceDesc object can be used to obtain information about this service description which your Service can use to dynamically alter the behavior of the service.

StartupListener

A StartupListener can be registered with the ServiceManager so that it can be notified about the boot process taking place. An example of such a StartupListener could be a SplashScreen service that listens for each Service Starting and displays a message showing which services are starting.

To install a StartupListener implement the **org.xito.boot.StartupListener** interface in a class and install the listener into the ServiceManager in the **initService** or **main** method of the service, using this method:

ServiceManager.getManager().addStartupListener(myListener);

Launcher API

The Launcher API can be used by a service, or any application running in BootStrap, to launch additional applications. Also any other utility API's provided by the BootStrap including reflectKit and Dialog Manager can be used by services or applications.

The BootStrap supplied launcher apis can launch any basic java application taking advantage of BootStrap Cache and Security Manager etc. A separate Launcher Service is available separate from the BootStrap that adds support for launching Java Applets, or JNLP (Web Start) applications.

An Executable Descriptor extending from ExectutableDesc is used to describe your application that is to be launched. **ExectutableDesc** is abstract, so BootStrap provides a concrete implementation called **AppDesc**.

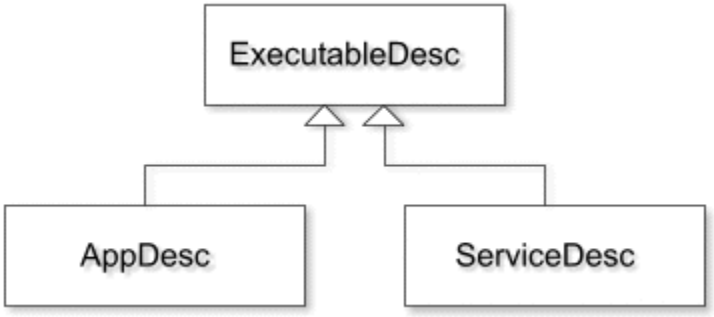


Illustration 4Class Diagram of ExecutableDesc

To launch an application in BootStrap you follow these steps:

1. Instantiate an AppDesc Object
2. Populate the AppDesc with information about your application: Jars, Security Requirements, Main Class etc.
3. User the Launcher class to execute your application

The following code shows an example of launching a simple application from within a BootStrap Service or application:

```
try {
  AppDesc appDesc = new AppDesc("Asteroids", "Asteroids");
  String codebase = "http://xito.sourceforge.net/apps/games/asteroids/";
  appDesc.addURL(new URL(codebase+"asteroids.jar"));
  appDesc.setPermissions(appDesc.getAllPermissions());
  appDesc.setMainClass("org.xito.asteroids.MainApp");
  appDesc.setNewAppContext(true);
  AppLauncher launcher = new AppLauncher();
  launcher.launchBackground(appDesc);
}
catch(Exception e) {
  //Show user error
}
```

The constructor of AppDesc takes two String parameters the first is the Application Name, the second is the Display name of the Application. These are used in security dialogs and error messages.

When specifying the permissions the application will require, you should currently use 1 of 2 options either **`appDesc.setPermissions(AppDesc.getAllPermissions())`** or **`appDesc.setPermissions(AppDesc.getRestrictedPermissions())`**

When using the AppLauncher class to launch the application described by the AppDesc you can call 1 of 3 methods they are described below:

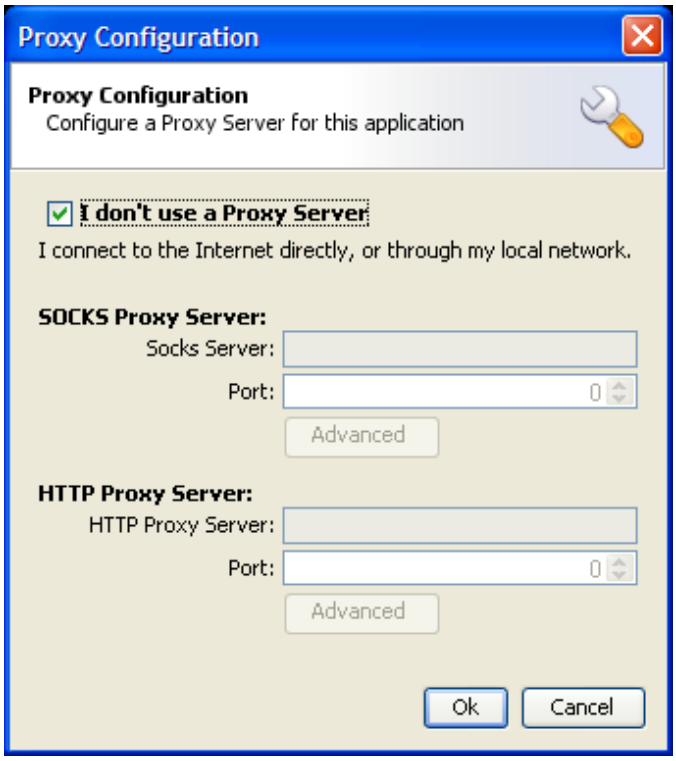
AppLauncher Method	***Description***
launchBackground(appDesc)	This will launch the application in a separate Thread in the Background. This method will not throw an exceptions if an execption is thrown during execution the BootStrap will display an error Message.
launchInternal(appDesc)	This will launch the application in a separate thread by the calling Thread will join this new Thread causing the callling application to block until the execution is complete. This method will throw an Exception if there is a problem launching the application
launchExternal(appDesc)	This will launch the application in a new VM and BootStrap. A new VM will be launched the BootStrap will be launched in minimum mode and the application will be started.

Proxy Server Configuration

Because BootStrap is designed to run applications over the Internet or local network the configuration of Proxy settings by users that need them becomes an important aspect of the user interface. Bootstrap provides built in proxy configuration for your application environment so you don't need to worry about it for your applications.

The first time the BootStrap environment is executed BootStrap checks to see if a **proxy.properties** file exists in the app.base.dir. If the file exists Bootstrap will load the settings in the file and continue booting the services. If the file does not exist and if the **boot.proxy.prompt** is set to **true** (the default), BootStrap will prompt the user for their proxy settings.

The following dialog shows the options available:



This Proxy Configuration is a thin wrapper around the Java Proxy implementation. Java uses a set of System.properties that determine the proxy configuration. If the user specifies Proxy Settings in the dialog BootStrap will write those settings into the required System.properties.

The default setting is just set to: “I don't use a Proxy Server” in this case the user simple hits OK and the environment continues to boot. If the user hits Cancel the environment will boot but they will be prompted again next time the environment is launched.

If the boot.propeties setting of **boot.proxy.prompt** is set to **false** or the boot.properties file contains proxy settings hard coded in it the proxy prompt will not appear.

Note: Any application that has permission can cause the Proxy Prompt to reappear by calling the **org.xito.boot.ProxyConfig.getProxyConfig().showProxyDialog();**

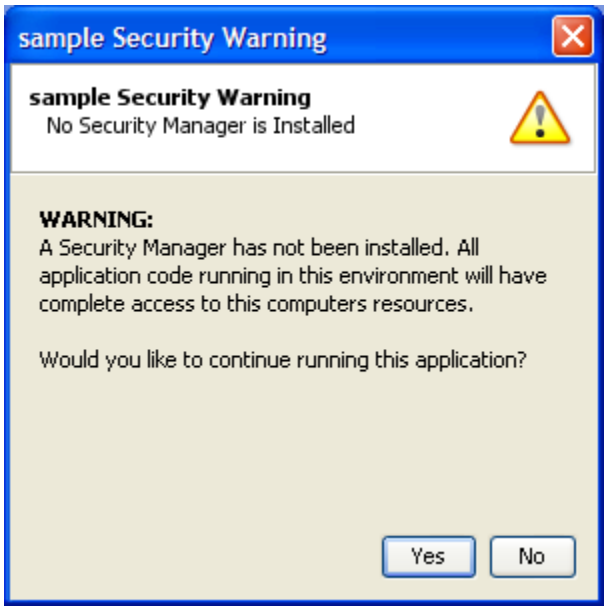
Security Configuration

Java Security is one of the most powerful features of the Java Architecture and has been pushed to the sideline by most Java Developers. Often Java Developers dislike Security Managers and quickly try to determine ways to turn them off so that the do not need to worry about security concerns in their applications. In general this is a mistake. Java Security is the developers friend and will make your applications safer and your users accept your applications easier.

The Security Manager and user interface for the security policies provided in Xito Bootstrap are designed to be easy to use by end users and enhance your applications, not hinder them. For this reason it is **highly recommended** that if you are using BootStrap to boot your own application environment you not disable the security manager.

That said, it is possible to disable the built in BootSecurityManager. The only valid reason for doing this is if you plan on implementing your own Security Manager in one of the boot services.

To disable the built in security you can specify: **boot.no.security=true** in the **boot.properties** file. However if you take this option and then fail to install a Security Manager in one of the Boot Services the BootStrap will warn your users that there is no Security Manager installed with the following Alert Dialog:



Because of these restrictions the only way to run BootStrap based applications without a security warning is to disable the built-in security manager in boot.properties and then implement and install your own Security Manager. In summary, it it best to work in the boundaries of BootStrap's default security environment.

Security Policy

Java Security Functionality is out of the scope of this document. Sun has published many documents concerning Java Security Architecture and these documents should be consulted for an in depth understanding of Security features in Java.

Xito Bootstrap attempts to make Security easy to understand for developers deploying applications on top of the bootstrap.

In general the following Security Policy rules are implemented in the BootStrap:

1. All Permissions specified in Java's default java.policy file are honored.
2. All Code that resides in the boot directory of bootstrap are granted All permissions.
3. All Other code that requests all Permissions will require the User to Grant those permissions.
4. Code that has not be granted All Permissions will run in a Restricted Permission Mode.

It is envisioned that the majority of service and application code running on top of BootStrap will be downloaded and executed from the Internet or some other local network server. **Therefore, the vast majority of application code running within the BootStrap environment will fall under rule: 3 or 4 above.**

The easiest way to ensure that your application has All Permissions without causing the Security Prompt to occur is to place your code in the boot dir of the BootStrap. This will cause the code to be run under rule 2 of the policy.

The assumption with granting all code in the boot dir with All Permissions is that the user has gone through the work to download and install your application “bootstrap” onto their computer. Therefore code that has been bundled along with bootstrap and placed in the boot dir should also have the same permissions that bootstrap itself enjoys. Of course deploying your application in such a way requires that users download new versions of your bootstrap based app each time you make changes. Having bootstrap launch your application over the net removes this issue, and is the preferred way to use BootStrap with your applications.

Security Permission Prompt

If your application or service code resides in any location other then the boot dir the BootStrap Security Manager will prompt your users to grant permission for this code to run.

The Permission prompt has the following information:

Description: This is basically a sentence that describes what the user is being asked to do. An example description is:

Would you like to Grant: **(All Permissions)** to code source: **http://host/.../myapp.jar** for "My Application"

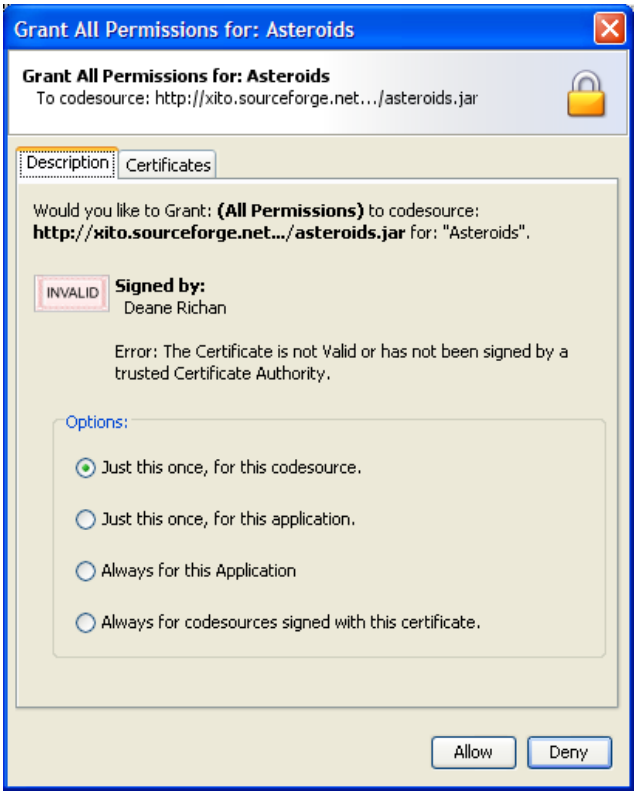
Signed By: If the application code has been signed the prompt will display who it is signed by and whether the certificate used to sign the code is valid. Valid certificates are not expired and signed by a valid root certificate authority. Detailed information about the certificate can be found by clicking view certificate on the security prompt dialog.

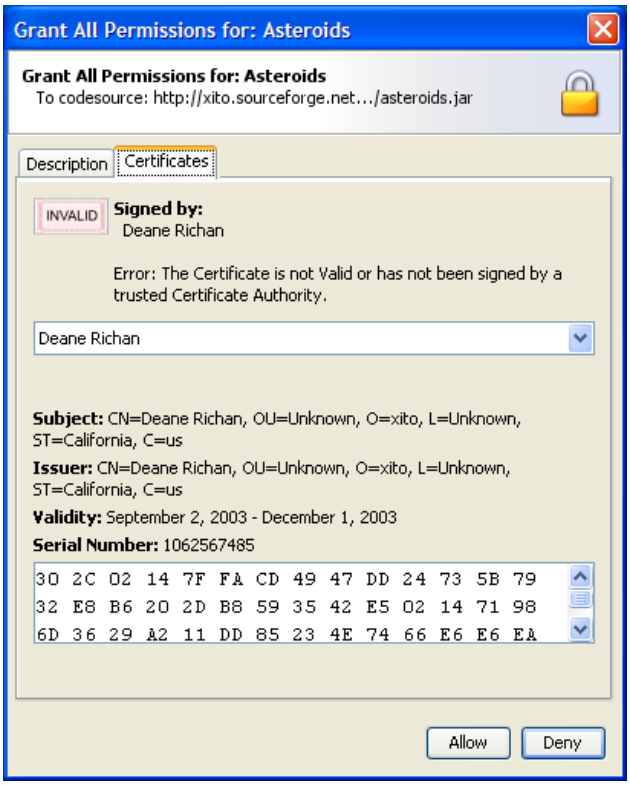
Grant Options: The user will be given the following grant options:

Option	***Description***
Just this once, for this code source	Grant permissions to the code source for just this Execution.
Just this once, for this application	Grant permissions to this code source and other code sources for this application for just this Execution
Always, for this application	Grant permissions to this code source and other code sources for this application for this and all future sessions
Always, for code sources signed with this certificate	Grant permissions to this code source and any other code sources signed by this certificate for this and all future sessions

A dialog showing an example of a security prompt along with an example of the Certificate Tab is shown below. The Certificate information can be used by users to obtain more information about the certificates used to sign this application's code and whether they are valid certificates or not.

If the user chooses **Allow** the security permission will be granted with the option they specified. If the user chooses **Deny** the application will be given Restricted Permissions an application should be designed so that it can run with Restricted Permissions or in the least case inform the user that it can't run in restricted permissions and exit.





Restricted Permissions

When applications are run in the Restricted environment they are granted only a handful of permissions. These are basically the same permissions that un-signed applets receive when they are running on a web page. The following table lists the permissions that are granted in Restricted Mode:

Permission	***Implication***
java.net.SocketPermission("localhost:1024-", "listen")	The application can listen to any Socket on the localhost on port 1024-65535
RuntimePermission("stopThread")	The application can stop threads it creates

Permission	*Implication*
PropertyPermission("java.version", "read") PropertyPermission("java.vendor", "read") PropertyPermission("java.vendor.url", "read") PropertyPermission("java.class.version", "read") PropertyPermission("os.name", "read") PropertyPermission("os.version", "read") PropertyPermission("os.acrch", "read") PropertyPermission("file.separator", "read") PropertyPermission("path.separator", "read") PropertyPermission("line.separator", "read") PropertyPermission("java.specification.version", "read") PropertyPermission("java.specification.vendor", "read") PropertyPermission("java.specification.name", "read") PropertyPermission("java.vm.specification.version", "read") PropertyPermission("java.vm.specification.vendor", "read") PropertyPermission("java.vm.specification.name", "read") PropertyPermission("java.vm.version", "read") PropertyPermission("java.vm.vendor", "read") PropertyPermission("java.vm.name", "read")	The application can Read all these System Properties
SocketPermission([CodeSource Host], "connect,accept")	Application can make TCP/IP connections to the host its code was downloaded from.
FilePermission(CodeSource, "read")	Application can read contents out of its own code source jar.

All other Java API calls that require additional permissions such as accessing the local hard drive, printing, connecting to your local network servers, connecting to random Internet servers, or even exiting the VM will not be allowed and will cause a SecurityException to be thrown. Well behaved applications should detect these exceptions and enable their applications to run in a crippled mode or Display a warning to the user and exit.

One of the Benefits of the Xito Bootstrap is its user friendly implementation of this security environment making it possible for many applications to run in restricted mode without a being a burden to the user. **In fact application designers should consider designing their applications to work in a restricted environment allowing end users the assurance that the application won't do anything it is not suppose to.**

Note: The separate Launcher Service provides the JNLP implementation of JNLP Services that enable restricted applications to access the File System, Printer Services etc in a controlled fashion. Your application can use these services to can access to local resources without requesting your application have All Permissions.

Executable Descriptors and Security

In order for BootStrap to launch a Service or Application an Executable Descriptor Object must be created. This object defines information about the application to be executed. Values for Application Name, Code sources, and Permissions can be specified.

During execution the Security Manager uses these Executable Descriptors to determine what permissions the Application wants to be granted. The SecurityManager will call **getPermissions()** on the ExecutableDesc class.

If the permissions returned from this imply the All Permissions then the security prompt will be displayed.

The BootStrap implements two versions of the ExecutableDesc: AppDesc and ServiceDesc

These Descriptors are used to launch applications and services. Thje separate Launcher Service provides Executable Descriptors for Applets and JNLP applications.

Services default to requesting All Permissions. Applications default to requesting Restricted Permissions. If you have application or service code that would like to launch another application with All Permissions the following code example can be used:

```
AppLauncher launcher = new AppLauncher();
try {
  AppDesc appDesc = new AppDesc("Asteroids", "Asteroids");
  String codebase = "http://xito.sourceforge.net/apps/games/asteroids/";
  appDesc.addClassPathEntry(new ClassPathEntry(new
URL(codebase+"asteroids.jar")));
  appDesc.setPermissions(appDesc.getAllPermissions());
  appDesc.setMainClass("org.xito.asteroids.MainApp");
  launcher.launch(appDesc);
}
catch(Exception e) {
  //Show user error
}
```

Sharing Virtual Machine Issues

Xito BootStrap is designed to launch multiple applications in the same VM. Although in practice the implementation works for the most part there are various issues that developers should be made aware of while running in a Shared VM.

System.exit

Traditionally Java developers assume their applications are the only applications running in a VM. Therefore their application will often create many Frames or Windows, spawn several Threads and then when the user wants to exit the application the developer simple calls System.exit(0).

On the surface this seems reasonable but in a shared VM world this behavior is not desired.

In order to allow multiple applications to run in a single VM BootStrap SecurityManager enforces that only a Single Class has the permission to exit the VM. By default this class in the Boot class.

Because of this security implementation when any other application running in the shared VM calls System.exit() a security exception will be thrown.

In general this approach is reasonable except when an application is running that does not know that it is running in the BootStrap environment ie (most current applications). For these applications the user will choose Exit from the UI and then the app will call System.exit() a security exception will be thrown and the application would still be running. Very frustrating to the end user. Because of this when such an application attempts to call System.exit the SecurityManager will attempt to shutdown their application through other means. It does this by keeping track of all Windows and Frames the App has created and then calls dispose on each one. It also attempts to keep track of all Threads started by the application and call stop on each one.

This approach is problematic especially the Thread.stop issue because this method has been deprecated for being DeadLock prone.

Although this mechanism works for the most part it is much better if applications avoid calling System.exit(). Rather then call System.exit applications should simply dispose of their own windows and notify any running threads to die on their own. This way their application will stop running without calling System.exit().

This is the best approach to take until Sun increases the ability of Java Applications to run in a single VM or multiple VMs with low overhead.

As part of the Xito Bootstrap a special class called AppShutdownHelper has been created to facilitate shutting down your application in Xito. This can be used with reflection so that your application will perform the same while running in non Xito platforms but will shutdown your application in Xito without calling System.exit.

In place of System.exit use the following code sample:

```
try {
   Class.forName("org.xito.boot.AppShutdownHelper").newInstance();
}
catch(Exception exp) {
   System.exit(0);
}
```

Simpling instantiating an AppShutdownHelper anywhere in your application code will cause your application to be destroyed.

Shared Virtual Machine Resources

There are many parts of the virtual machine that multiple application must share. If your application makes changes to these single resources other applications in the environment may suffer. Some examples are System.properties, Security Policy, Security Manager, java.awt.Toolkit, URLStreamHandlerFactory and other VM Singletons. If your application needs to change or manipulate these or other VM shared resources it should probably not run in a shared environment. It should either find other ways to implement the functionality in a spirit of cooperation or simply choose to run in a separate VM.