

dotFX Version 1.2.7

User Guide

©2009 dotFX, Inc. All rights reserved.

1 Introduction

dotFX is a framework for running Java applications over networks. Version 1.2.7 offers the following features:

- It allows the application programmer to write networked applications without doing any extra work. In particular, the programmer does not have to write networking code, generate proxies or stubs, create extra interfaces, or use a directory service. The application may, in fact, be written and tested as a stand-alone, then simply dropped into the dotFX framework where it will immediately gain networking capabilities.
- It runs on a true object model, rather than XML data structures and stateless procedures.
- It enables fully functional software to be launched via a clickable link.
- It provides automatic versioning (and transparent update) of individual classes, resources, JVMs, and JREs at runtime.
- It can run on-line or off-line.
- It provides client-side security, which makes it possible for the client safely to run untrusted code.
- It provides all of its basic capabilities without an a.p.i. This means that virtually any application that runs in the dotFX framework will also run outside it as a stand-alone.

2 Installation

2.1 Client

Download the appropriate installer from <http://www.dotfx.com/>, click it, and follow the instructions.

2.2 Server

Mac, Linux, BSD:

1. Download the appropriate tar file for your system from:
<http://www.dotfx.com/>
2. Select an installation directory and untar the file. The dotFX installation will go into a “dotFX” directory under the current directory.
3. Follow the instructions in the README file in the dotFX directory.

Windows:

1. Download the appropriate installer image from:
<http://downloads.dotfx.com/>
2. Click on the installer and follow the instructions.

3 Quick Start

The dotFX framework is designed to make the writing and distribution of networked applications extremely easy. The basic steps are:

1. Install a dotFX server (see section 2.2).
2. Write your application in Java and compile it.
3. Copy the application files (including all libraries) into the server’s repository (see section 5) as a directory tree, as JAR files (see section 9), or as some combination of both. If you want the application to have its own private area of the server repository (unshared with other applications), see section 6.
4. Start the dotFX server by following the directions in the README file located in the dotFX installation directory.

Your application is now accessible from any dotFX-enabled client machine through a URL of the form “fx://myserver.com/my.app.MainClass” (or “fxs://myserver.com/my.app.MainClass”). In many cases, this is all there is to it.

Certain applications, however, may encounter security constraints due to the fact that the dotFX client runs in a security sandbox (see section 16 for details). The most common security-related issues are attempts to read or write the client’s disk outside the authorized area of the repository and attempts to open network connections to machines other than the server hosting the application.

Rather than reading or writing the user’s home directory, applications should use their own area of the client repository. The path to the client repository is stored in the Java system property “com.dotfx.working.dir”.

If the application needs to open network connections to arbitrary machines, it can do so by means of a server class (see section 10).

A privileged mode (see section 16.2) is available for applications requiring full user privileges.

4 dotFX Framework Processing Flow

The logical model of the dotFX framework consists of a client that runs applications locally (at least in part), and a server that supplies application classes and resources to clients, and can also run parts of client applications. Each running application on the client owns a stateful session with the server, though the session may be short lived (for example, it may last only long enough for the client to download the application code). An application class may also be *instantiated* on the server, in which case it lives until the session ends, the object passes out of context and is finalized, or the object is explicitly released by the client. Each session runs in its own name space.

The general processing flow for a dotFX application is as follows:

1. A URL of the form “fx://heliox.com:8080/com.heliox.MainApp” is passed to the dotFX client. If the port number is omitted, it defaults to 80 for the “fx://” protocol, and 443 for the “fxs://” protocol (which uses SSL). The URL may additionally include some configuration parameters indicating how the application should run (see section 15).
2. The client attempts to locate com.heliox.MainApp in the heliox.com area of its repository. If the class is found, the client reads MainApp’s version number or checksum (see section 7).
3. The client connects to heliox.com and sends its version number (if any) or checksum for MainApp. If the server has a newer version, the client loads it; otherwise, the client loads the version from its repository.
4. Classes and resources used by MainApp are loaded from the client repository or from heliox.com, as in step 3. If a loaded class is tagged as a server class (see section 7), any instances of that class created by the application will exist on the server, and method invocations (including statics) will be transparently proxied.
5. MainApp runs on the client machine under the dotFX security manager, while any server objects employed by MainApp run on the server.

5 The dotFX User Directory

The dotFX user directory contains log files, client and server repositories, keys, databases, and most of the dotFX framework itself. It is also the default location for the dotFX properties files.

There should be one user directory per user, and its location is set by the “FXUSERDIR” Java property. If this property is not set, then the user directory will default to the following system-dependent location:

Mac OS X: `${HOME}/Library/Application Support/dotFX`

FreeBSD, Linux, Solaris: `${HOME}/.dotFX`

Windows: `${HOME}\Local Settings\Application Data\dotFX`

The user directory contains the following sub-directories:

- **db**

This directory contains performance-optimization databases for the server. The contents of the directory are automatically generated by the server, and it is safe to delete them while the server is not running. Such deletion is recommended if the server’s repository has undergone substantial revision. This may slightly improve efficiency by eliminating database entries corresponding to items that no longer exist.

- **fx_m.n**

where “m” and “n” are integers. This directory is for internal framework use.

- **jre_m.n**

where “m” and “n” are integers. This directory is for internal framework use.

- **keys**

This directory contains SSL certificates and keys.

- **log**

This directory contains framework-generated log files.

- **repository/server**

The contents of this directory are structured like a Java class path, and comprise the set of Java classes and other resources served by the dotFX server. Nothing outside this directory is accessible for download through the server.

Note: the repository *must not* be in the server’s class path.

- **repository/server/DOTFX**

This directory is for internal framework use.

- **repository/server/FXJRE**

This directory is for internal framework use.

- **repository/client/normal**

This directory contains a set of sub-directories. The name of each sub-directory is the same as a host name from which the client has loaded an application via the “fx://” protocol, and the contents are structured like a Java class path. Each sub-directory contains classes and other resources that were cached or created by applications invoked from one particular host. For example, applications invoked from `fx://heliox.com/` will cache themselves in `repository/client/normal/heliox.com`. Such applications are free to read from and write to any area under this directory, but nowhere else. Applications may obtain the path to this directory by calling

```
System.getProperty("com.dotfx.working.dir")
```

- **repository/client/secure**

This directory performs the same function as `repository/client/normal`, except that it contains code and resources for applications loaded via the “fxs://” (SSL) protocol.

All other directories are for internal framework use and should not be altered.

6 Application Domains

An application domain is simply a sub-directory under the server’s repository (see section 5) that acts as a self-contained root for one or more applications. When loading an application in ClientFX, the URI should specify the domain (if any) in its path. For example:

```
fx://apps.heliox.com/mydomain/test.MyApp
```

will load the main class “MyApp” from package “test” in application domain “mydomain.” This means that the server will expect to find MyApp in

```
FXUSERDIR/repository/server/mydomain/test
```

Code and resources in an application domain are not shared by other domains. It is therefore safe to use different versions of the same class or library in different domains.

Application domains can be aliased by making an entry in the server’s properties file (see section 13.2).

7 FX Tags

Classes loaded via the dotFX client may contain certain tags that affect how the framework handles them. Tags are inserted as static variables in the class to which they pertain, and the actual value and type of the variable are irrelevant (only the variable name is read). For example:

```
static byte fxServerClass;
```

indicates that the class in which this tag is declared is a “server class.” A server class is a class that gets instantiated on the server, rather than the client. See section 10 for more details.

Other useful tags are:

- **fxNeedsPrivilege**
Causes the user to be queried as to whether or not the application should be allowed to run with expanded privileges. See section 16.2.
- **fxDeprivilege**
Causes a previously privileged application to have its privileges revoked. See section 16.2.
- **fxClassVersionID_x_y_z**
The class has version “x_y_z,” where x, y, and z are any integers (the version i.d. may actually contain any number of integers, each separated by an underscore). This version i.d. will be used when deciding whether or not a client’s cached version of a given class should be superceded by the server’s version. Version i.d.s are compared in the usual way.
- **fxClassVersionID_force_update**
When this tag is present in *either* the client’s cached version of the class or the server’s version, the client will load the server’s version.
- **fxUpdateController**
This tag may be declared in a main program class to indicate that classes used by this application will be updated from the server if the main class itself has been updated from the server, and not otherwise. In other words, the tag slaves versioning for the entire application to the main application class.

If a class does not have an explicit version number and is not slaved to an `fxUpdateController`, its version is determined by its checksum. If the client’s checksum for the class is different from the server’s, then the client will load the server’s version.

8 Application Resources

The dotFX framework will find an application resource (loaded through the standard `ClassLoader.getResource()` mechanism) whether the resource resides on the client or the server. If the resource exists in the client’s repository, a new copy of the resource will be downloaded only if the server’s version of the resource is different. Resource files are compared by means of a CRC checksum.

9 JAR Files

In many cases, the easiest and best way to deal with a JAR file is to extract its contents into the server's repository. The dotFX framework can also handle JAR files directly. Simply place your JAR files at the root of your server repository or application domain (see section 6), as appropriate. Any such JAR file is on the application's class path, and will automatically download to the client when any of its contents are needed. Each JAR file is handled as a unit, so the whole file transfers to the client at once. Note that tags (see section 7) are not supported in classes loaded from JAR files.

10 Server Classes and Network Connections

A server class is a class whose instances exist on the server and whose methods execute on the server (irrespective of where those methods are invoked). Server classes are implemented through use of the “`fxServerClass`” tag (see section 7). They must extend either `java.lang.Object` or another server class and have a no-argument constructor, and they differ in the following ways from ordinary classes:

- Any parameters passed into server objects from client-side objects or returned from server objects to client-side objects are passed by value instead of reference, and consequently must be `Serializable`. The exception to this rule is server objects themselves, which are always references (to objects on the server) irrespective of where they are passed.
- Final methods of server-class super-classes are not accessible from outside the server.
- Data members of server classes are not directly accessible from outside the class (though they are, of course, accessible via getter and setter methods). This is simply good programming practice.

For an example of a server class, see appendix A.1.

The server's properties file controls the number of instances of any server class that can exist globally at one time (see section 13.2.1). Every server class must be enabled in the properties file before it can be instantiated. Static methods on server classes can only be invoked if at least one instance of the class is allowed for instantiation (in this case, the class need not have a constructor; it only needs to be allowed in the properties file).

Application code running on the server (i.e., invoked from a server class) resides in a separate name space for each session. Code that needs to be in a global (server-wide) name space should be placed in a common code directory specified by the server's `commonCodePath` property (see section 13.2).

Every instance of a server class has an implicit “`public void _DESTROY()`” method that allows the client to release the instance explicitly (the class may

define this method with an empty body, and it will be implemented by the framework). If this method is not called, the instance is released when the session ends or when the object is finalized on the client.

The dotFX framework maintains one connection to the server for each thread in which a server class is instantiated or has a method invoked. These connections are opened transparently as they are needed, and use whichever protocol was specified in the URL used to start the client. Multiple server objects in a single client-side thread share the same connection. A server class may also declare a “`public void _CLOSE(Thread t)`” (or “`public static void _CLOSESTATIC(Thread t)`”) method (again, with an empty body) that will explicitly close the network connection to the instance’s host for a particular thread. The current session ends (and server objects are released) when all application connections are closed. In other words, the session will persist as long as at least one connection is open. If a connection is not explicitly closed, it will close when its associated thread is finalized.

In all other respects server objects behave the same way as ordinary objects. They belong to the same inheritance chain, can implement static methods, can be passed as parameters to client-side objects or to other server objects, etc.

In addition to network connections used by the client application, the dotFX framework itself may use one or more network connections for the purposes of class and resource loading. These connections time out when not in use.

11 Server Common Path

The server’s common path (specified through the `commonCodePath` property; see section 13.2) is a subdirectory under the server repository. The contents of this area have three important characteristics:

1. They are globally accessible to any code running on the server.
2. They run in a server-wide name space.
3. They are not accessible to any client.

These characteristics impart the ability for different application instances on the same server to communicate with one another (recall that code belonging to an application instance normally runs in its own name space). For example, an application could use a common-path singleton to dispatch chat messages among separate running sessions, or to share a database connection.

Some caution is necessary when using the common path due to the fact that the client side of the application does not have access to any of its contents. While it is generally safe to ensure that common code and resources are only loaded behind a server-class (see section 10), surprises can occur if, for example, a common-path object throws an instance of an exception class that the client cannot resolve. If the client needs access to common code, such code may be copied into the application’s domain (see section 6). In this way, the server

side of the application will load the code from the common path (because the common path has precedence in the server's search order), while the client will be supplied from the application domain. This is safe, as long as the class definitions match.

12 Offline Applications

An application can be configured to run offline by taking the following steps:

1. Install the application in its own application domain (see section 6).
2. Declare the application's domain in the `offlineAppDomains` property in the `serverfx.properties` file. See section 13.2.
3. Include "offlineable=true" in the URI query parameters when invoking the application from a browser or other client (see section 15 for information on query parameters).

For example, if the application `MyApp` with main class `MyApp.class` is installed in the application domain `MyAppDomain` on host `apps.dotfx.com`, then the application can be enabled to run offline by including the line

```
offlineAppDomains=MyAppDomain
```

in the `serverfx.properties` file and loading

```
fx://apps.dotfx.com/MyAppDomain/MyApp?offlineable=true
```

in a client.

Offline applications are provisioned and updated incrementally and in the background (so as not to delay the user's work with the application). Incremental re-provisioning occurs whenever the application's server is available and the application's code or resources have changed. The user is notified before and after any provisioning or re-provisioning takes place, and may decline to re-provision.

An offline application only runs offline when its server is unavailable, so it is otherwise free to use server classes (see section 10) and other online resources. In this case, however, the programmer is advised to catch runtime exceptions from server classes so that the application will behave gracefully when disconnected from the network.

13 Framework Properties

13.1 Client Properties

The client's properties file may be specified by setting the Java property "FX-CLIENTPROPS". The default properties file is `FXUSERDIR/clientfx.properties`.

If no properties file exists when the client starts up, one will be created in the default location.

The client recognizes the following properties:

- **cacheEnabled**

This property indicates whether or not the client will cache loaded application classes and resources. The default value is true.

- **clientLogFile**

This property specifies the name of the client-generated log file. Log files are kept in `FXUSERDIR/log`. The default file name is `clientFX-n.log`, where `n` is an integer that increments when the log file rolls over.

- **sslTrustStorePass**

This property specifies the password for reading the client's SSL trust store. The default value is "123456". Note that the trust store contains only public certificate information and consequently does not need to be secret.

- **sslTrustStorePath**

This property specifies the client's SSL trust store. The default is `FXUSERDIR/keys/truststore`.

- **useNetwork**

This property indicates whether or not the client uses the network. If set to false, all application code and resources must be available in the client repository. The default value is true.

13.2 Server Properties

The server's properties file may be specified by setting the Java property "FX-SERVERPROPS". The default properties file is `FXUSERDIR/serverfx.properties`. If no properties file exists when the server starts up, one will be created in the default location. The server installation includes a sample properties file called `serverfx.properties.default`, in the main installation directory.

The server recognizes the following properties:

- **autoReload**

This property specifies the interval (in seconds) at which the server reloads any reloadable properties from its properties file.

- **commonCodePath**

This property specifies a class path (relative to `FXUSERDIR`) for code that will go into a server-wide name space when loaded by the server (see section 11). There is no default.

- **maxConnections**

This property specifies the maximum number of concurrent connections the server will accept. The default value is infinity (indicated by “maxConnections=*”). It is generally best to keep the default and to use the maxSessions property for load-limiting purposes.

- **maxSessions**

This property specifies the maximum number of concurrent sessions the server will accept. A session is defined as one invocation of an application for one user, and may employ any number of connections. The default value is infinity (indicated by “maxSessions=*”).

- **offlineAppDomains**

This property contains a colon-separated list of application domains (relative to FXUSERDIR/repository/server) whose contents are scanned on startup. The purpose of this is to update the server’s database so that applications in these domains are available for offline use (see section 12).

- **serverLogFile**

This property specifies the name of the server-generated log file. Log files are kept in FXUSERDIR/log. The default file name is serverFX-n.log, where n is an integer that increments when the log file rolls over.

- **serverPort**

This property specifies the port number on which the server listens for normal incoming connections (using the “fx://” protocol). There is no default.

- **serverSslPort**

This property specifies the port number on which the server listens for SSL-encrypted incoming connections (using the “fxs://” protocol). There is no default.

- **sslKeyStorePass**

This property specifies the password for the server’s SSL key store. There is no default value.

- **sslKeyStorePath**

This property specifies the server’s SSL key store relative to FXUSERDIR. There is no default value.

- **useServerDB**

This property indicates whether or not the server should use a database for performance optimization. The default value is true. This property must be set to true if there are any offlineAppDomains.

- **serverObjectLimit.adminfx/com.dotfx.adminfx.AdminFXAgent**

This property and the following indicate the number of allowed simultaneous instances of the AdminFX server monitor (see section 18). The value set for these two properties should always be the same. The default value is 0.

- **serverObjectLimit.adminfx/com.dotfx.adminfx.ServerLogin**

See above.

13.2.1 Server-class Instances

A property of the form

```
serverObjectLimit.myAppDomain/my.fully.qualified.ClassName=n
```

where n is either an integer or "*" (the latter meaning "infinity") indicates that the server will instantiate at most n instances of myAppDomain/my.fully.qualified.ClassName at one time (globally). A class will not be available for instantiation on the server unless its name appears in this form with an associated value greater than zero. Invocation of a static method on the server requires that the defining class be available for instantiation (as defined in this paragraph), though in this case, the number of allowed instances is ignored as long as it is greater than zero. If a class is unavailable for instantiation (either because the maximum number of instances has already been created or the number of allowed instances is zero), the server returns a runtime exception. This property is regularly re-read by the running server, so it is unnecessary to reboot the server after adding, deleting, or modifying entries.

There is one special server whose accessibility is also controlled in this way:

- **serverObjectLimit.ClassServer**

This property specifies the number of class servers (i.e., the objects that supply class definitions and resources to clients) that can be instantiated globally at one time. One or more class server instances service each user session, and they are released after a period of inactivity. The default value of the property is zero, though the automatically generated properties file will set this to infinity ("*").

13.2.2 Application-domain Aliases

A property of the form "appDomainAlias.my/app/domain=other/app/domain" creates an application-domain alias. If such an entry is found in the server properties file for the host apps.dotfx.com, then any client connecting to the URI "fx://apps.dotfx.com/my/app/domain/AnyApp" will be transparently redirected to "fx://apps.dotfx.com/other/app/domain/AnyApp". Domain aliases are regularly re-read from the properties file by running servers, so it is unnecessary to restart the server when an alias changes. This creates a convenient way to upgrade or modify running applications without interrupting service.

14 Java Properties

The following Java properties (accessible through calls to `System.getProperty()`) are available to application code running on the client or the server:

- **com.dotfx.framework.role**
A value of “client” indicates that the calling code is executing in the dotFX client. A value of “server” indicates that the calling code is executing in the dotFX server.
- **com.dotfx.user.dir**
Contains the path to the dotFX user directory (see section 5).

The following properties are available to application code running on the client:

- **com.dotfx.working.dir**
Contains the path to the working directory for the current application (see section 5).
- **com.dotfx.current.dir**
Contains the path to the cache directory for the current application. This will usually be the same as the working directory, but may sometimes be a temporary scratch area. In practical terms, it is the common root directory for URLs returned by calls to `ClassLoader.getResource()` or `ClassLoader.getResources()`.

15 Query Parameters

ClientFX recognizes the following query parameters in the URI:

- **mem**
Sets the maximum Java heap size. The value of this parameter is passed to the JVM via `-Xmx`. For example, `mem=128M` starts the JVM with `-Xmx128M`.
- **meminit**
Sets the initial Java heap size. The value of this parameter is passed to the JVM via `-Xms`. For example, `meminit=16M` starts the JVM with `-Xms16M`.
- **memperm**
Sets the initial heap size for the Java permanent generation. The value of this parameter is passed to the JVM via `-XX:PermSize`. For example, `memperm=128M` starts the JVM with `-XX:PermSize=128M`.

- **mempermmax**

Sets the maximum heap size for the Java permanent generation. The value of this parameter is passed to the JVM via `-XX:MaxPermSize`. For example, `mempermmax=160M` starts the JVM with `-XX:MaxPermSize=160M`.

- **backgroundonly**

If set to “true,” this parameter sets the “`java.awt.headless`” property.

The following query parameters are recognized by Macintosh clients, and ignored by others:

- **appicon**

Sets the path to the dock icon for the application. The value of this parameter is passed to the JVM via `-Xdock:icon`. For example, `appicon=/path/to/myicon` starts the JVM with `-Xdock:icon=/path/to/myicon`.

- **appname**

Sets the name of the application in the menu bar and the dock. The value of this parameter is passed to the JVM via `-Xdock:name`. For example, `appname=MyApp` starts the JVM with `-Xdock:name=MyApp`. Note that a bug in Mac OS X 10.4 prevents the name from being set correctly in the dock. This bug was corrected in Mac OS X 10.5.

- **macmenus**

If set to “true” (which is the default), this parameter causes the application’s menu to appear in the menu bar; if set to “false,” the menu appears in the application window. The value of this parameter is passed to the JVM via `-Dapple.laf.useScreenMenuBar`. For example, `macmenus=false` starts the JVM with `-Dapple.laf.useScreenMenuBar=false`.

- **offlineable**

If set to “true” (and the server agrees) this application can run offline (see section 12).

16 Security

16.1 Standard Privileges

Applications loaded by the dotFX client (from the local repository or the network) have the following default privileges:

- read and write to the area of the client repository owned by the server from which the application was loaded
- read and write to the system’s temporary directory

- open socket connections back to the originating server
- access the clipboard
- access the AWT event queue
- change the keyboard focus manager
- monitor the mouse cursor
- set a window to remain on top
- read the environment
- reflectively access declared members of classes (within normal accessibility constraints)
- queue print jobs
- get stack traces
- read and write Java properties
- play audio

Other actions requiring a security-manager check are not allowed. In particular, client code is not allowed to read or write the disk outside its own area of the repository, make socket connections to arbitrary addresses, or create class loaders.

16.2 Privilege Changes

Application privileges can be expanded on the client through use of the `fxNeedsPrivilege` tag (see section 7 for a general discussion of tags). When the client encounters this tag, it presents a query dialog to the user, asking if the user wishes to grant the application expanded privileges, and whether or not the choice should be made permanent. If the user agrees, security checks for the application are suspended, and the application effectively runs with full user privileges; if the user disagrees, the application continues to run with default (restricted) privileges.

The reciprocal of the `fxNeedsPrivilege` tag is `fxDeprivilege`. This tag quietly revokes expanded privileges (if any) for the running application and its future invocations.

Privilege expansion and revocation can happen only when the application is served over an SSL connection via the “fxs” protocol. Privilege expansion further requires that the server use a special “privileged” certificate issued by dotFX, though the certificate for host “127.0.0.1” (included in the ServerFX distribution) is privileged for testing purposes.

17 Native Code

Privileged applications (see section 16.2) are free to use native code. Simply place the native library in the root of the server repository or application domain (see section 6), and download the library to the client by means of a `ClassLoader.getResource()` call before the library is used. See appendix A.4 for an example.

18 The AdminFX Server Monitor

The server monitor allows system administrators to view the status of a running dotFX server. It provides several types of information about current sessions including IP addresses, data rates, instances of server objects, and applications being run.

Access to the server monitor is controlled through two server properties (`serverObjectLimit.adminfx/com.dotfx.adminfx.AdminFXAgent` and `serverObjectLimit.adminfx/com.dotfx.adminfx.ServerLogin`; see section 13.2) and a password file located at `FXUSERDIR/adminfx.password`. The password file simply consists of one or more lines of the form “username=password”.

The server monitor may be invoked at:

```
fxs://my.server.name/adminfx/com.dotfx.adminfx.AdminFX
```

Note that access to the monitor requires the use of SSL (as indicated by the “fxs” protocol specifier). The server distribution includes an SSL certificate for the host “127.0.0.1,” so you can run the monitor on the local machine (after enabling SSL in the `server.properties` file) with:

```
fxs://127.0.0.1/adminfx/com.dotfx.adminfx.AdminFX
```

If you need to run the monitor from a remote machine, please contact dotFX to obtain a server certificate.

A Examples

The code excerpts in this appendix are intended as uncluttered illustrations of various dotFX mechanisms, not as working examples. For more complete examples, please refer to the dotFX Web site.

A.1 Server Classes

The primary intent of the dotFX framework is to make it *very* easy to write and distribute networked applications. Here is a simple example that takes a string argument on its command line, sends the string to the server, converts the string to upper case on the server, then sends back and prints the result. The application consists of two classes, A and B.

```
public class A
{
    public static void main (String[] args)
    {
        B b = new B();
        System.out.println(b.go(args[0]));
    }
}

public class B
{
    static byte fxServerClass; // run this class on the server

    public B() {}

    public String go(String s)
    {
        return s.toUpperCase();
    }
}
```

After the compiled class files (A.class and B.class) are dropped into fx.heliox.com's server repository (FXUSERDIR/repository/server by default) and access to B is enabled by adding "serverObjectLimit.B=*" to the server's properties file (FXUSERDIR/serverfx.properties by default; see section 13.2.1), the application is accessible from anywhere on the Internet by entering or clicking on "fx://fx.heliox.com/A" from a dotFX-enabled client machine. It's that simple.

When the client invokes the application, class A runs on the client machine and class B runs on the server. All communications between the two are handled internally by the framework. Note also that if we take this application out of the framework, it will run just fine as a stand-alone. This makes it possible to do rapid development by coding and debugging applications as stand-alones, then dropping them into the framework when they're ready.

In the previous example, the instance of class B running on the server is released when the instance of B on the client is finalized (or when the application ends). If we want to release B explicitly, we do this:

```
public class A
{
    public static void main (String[] args)
    {
        B b = new B();
        System.out.println(b.go(args[0]));
        b._DESTROY(); // release B instance on the server
    }
}
```

```
public class B
{
    static byte fxServerClass; // run this class on the server

    public B() {}

    public String go(String s)
    {
        return s.toUpperCase();
    }

    public void _DESTROY() {} // framework will define this
}
```

If any method is invoked on the instance of B after `_DESTROY()` has been called, the framework will throw a runtime exception.

A.2 Versioning

If we wish, we can add explicit versioning to our application (as opposed to simple, checksum-based versioning, which is the default) by including one additional tag in each class to be version-controlled:

```
public class A
{
    static byte fxClassVersionID_1_0; // this is version 1_0

    public static void main (String[] args)
    {
        B b = new B();
        System.out.println(b.go(args[0]));
    }
}

public class B
{
    static byte fxClassVersionID_1_0; // this is version 1_0

    public B() {}

    public String go(String s)
    {
        return s.toUpperCase();
    }
}
```

The first time a client runs this application, it will be cached in the client's repository. Each time thereafter that the application is run, it will run from the client's repository (without downloading new code) unless a newer version of one

or more classes is available on the server. If, for example, the client has version 1_0 of class A and version 1_0 of class B in its repository and the server has version 1_1 of class A and version 1_0 of class B, the client will receive, cache, and use the new version of class A (1_1), but will use its previously cached version of class B (1_0). Server classes (classes tagged with “fxServerClass”) have no version number because they always live on the server. Note that version 1_0_0 is newer than version 1_0, and any version is newer than a class without a version tag.

It is also possible to force the client to use the server’s version of a class like this:

```
public class A
{
    static byte fxClassVersionID_force_update; // update from server
no matter what

    // ...
}
```

In this case, the client will get the server’s version whether the `fxClassVersionID_force_update` tag is found in the server’s *or* the client’s version of the class.

If we want to version-control the entire application as a unit, we do this:

```
public class A
{
    static byte fxUpdateController; // slave versioning to me
    static byte fxClassVersionID_1_0; // version 1_0

    private B b = new B();

    // ...
}
```

Here, class A and class B will both be updated from the server if and only if class A is updated. Any version tags in class B are ignored.

A.3 User Authentication

A simple way to perform user authentication is through use of a static variable on the server. For example:

```
public class Login
{
    static byte fxServerClass;
    private static boolean isLoggedIn;

    public Login() {} // required constructor
```

```
public static void doLogin(String uname, String pass)
{
    // Check the password file here.  If successful,
    // set isLoggedIn = true
}

public static boolean isLoggedIn()
{
    return isLoggedIn;
}
}

public class MyServerClass
{
    static byte fxServerClass;

    public MyServerClass() throws NotLoggedInException
    {
        if (!Login.isLoggedIn())
            throw new NotLoggedInException();
    }

    public void doStuff()
    {
        // ...
    }
}

public class MyMainClass
{
    public static void main(String[] args)
    {
        MyServerClass serverInstance = null;
        // Prompt user for name and password here.
        Login.doLogin(name, password);
        try { serverInstance = new MyServerClass(); }
        catch (NotLoggedInException e)
        {
            System.out.println("oops!");
            System.exit(-1);
        }
        serverInstance.doStuff();
    }
}
```

The idea here is that `MyMainClass` (running on the client machine) cannot obtain an instance of `MyServerClass` (running on the server) unless the user has correctly authenticated. Note that “`serverObjectLimit.Login`” and “`serverObjectLimit.MyServerClass`” must both be set in the `serverfx.properties` file (see section 13.2.1).

A.4 Native Code

Privileged applications (see section 16.2) can execute native code on the client. The following example shows how to load a native library appropriate to the client’s platform. It assumes that the files `libjogl.jnilib` and `jogl.dll` reside in the server’s repository at the root of A’s application domain (see section 6).

```
public class A
{
    static byte fxNeedsPrivilege; // request privileged execution

    public static void main(String[] args)
    {
        String osName = System.getProperty("os.name");

        if (osName.equals("Mac OS X"))
            A.class.getResource("/libjogl.jnilib");

        else if (osName.startsWith("Windows"))
            A.class.getResource("/jogl.dll");

        // ...
    }
}
```

The `jogl` library is now effectively on the client’s library path, and available for use by the application.

A.5 Streaming

The following example illustrates buffered data streaming from the server to the client. `ServerInputStream` must be enabled as a server class in the `serverfx.properties` file (see section 13.2.1).

```
public class ServerInputStream
{
    static byte fxServerClass; // run on the server
    private java.io.FileInputStream in;

    public ServerInputStream() {}

    public long setFile(String fileName) throws java.io.IOException
    {
```

```
        in = new java.io.FileInputStream(fileName);
        return new java.io.File(fileName).length();
    }

    public byte[] readFile() throws java.io.IOException
    {
        int read = 0;
        int total = 0;
        java.io.ByteArrayOutputStream out =
            new java.io.ByteArrayOutputStream();
        byte[] b = new byte[32768];

        while ((read = in.read(b, total, b.length - total)) > -1 &&
            (total += read) < b.length) {}

        out.write(b, 0, total);
        return out.toByteArray();
    }
}
```

```
public class ServerFileReader
{
    public static void main(String[] args) throws java.io.IOException
    {
        ServerInputStream sis = new ServerInputStream();
        long length = sis.setFile("/path/to/server/file");
        int bytesRead = 0;
        byte[] b;

        String workingDir = System.getProperty("com.dotfx.working.dir");

        java.io.OutputStream out =
            new java.io.FileOutputStream(workingDir + "/myfile");

        while (bytesRead < length)
        {
            bytesRead += (b = sis.readFile()).length;
            out.write(b);
        }
    }
}
```