Platform SDK Developer's Guide

Using the Warm Standby Application Block

4/3/2025

Contents

Using the Warm Standby Application Block

> **Tip**
>
> - This application block is a reusable production-quality component. It has been designed using industry best practices and provided with source code so it can be used "as is," extended, or tailored if you need to.
> Please see the License Agreement for details.
> - The Warm Standby Application Block described in this topic is a redesign first available with Platform SDK 8.5.101.06 for Java or 8.5.101.06 for .NET, that provides no backwards-compatibility for earlier releases. For information about earlier versions of the Warm Standby Application Block, please read the Legacy Warm Standby Application Block Description.

This article describes how developers can use the Warm Standby Application Block to maintain availability of connections between their applications and Genesys servers. It applies to all Server Deployment Modes, no matter if single-server mode, primary-backup mode, or cluster (N+1) mode.

The `WarmStandby` class is designed to handle the process of connecting (first-time connection) and reconnecting (in case of a connection failure) to the Genesys servers. `WarmStandby` maintains a pool of server addresses and sequentially tries to connect to a server until an attempt is successful or the pool has been exhausted.

`WarmStandby` raises events about its behavior that can be traced by client code.

Supported Server Deployment Modes

- **Single Server:** `WarmStandby` assists with reconnection attempts to the same server when the client gets disconnected.
- **Classical Genesys Primary-Backup:** `WarmStandby` assists with reconnection attempts to the same server, and failovers among the pair of servers. In some cases, failovers need to wait for a delay for the backup server to become active (for example, with Configuration Server).
- **Active N+1 Cluster:** `WarmStandby` assists with reconnections and failovers.

In addition, any of these configurations can be deployed in **Multiple Data Centers** mode. In this mode, the service is deployed across Data Centers, where every Data Center has a set of servers configured in any of the Server Deployment Modes above. `WarmStandby` enables the client to do Data Center failovers by allowing the pool of server addresses to be reconfigured. Triggering a Site failover (pool reconfiguration) is the responsibility of the client application, as it will depend on deployment-specifics.

Behavior

Behavior in the case of pool-exhaustion is defined programmatically by your client application, and

depends on your needs. In some cases, you will want WarmStandby to stop trying to connect. This is the case for client applications that need an open connection to go on with the application logic. In other cases you will want to continue attempting to connect, for keeping service availability. For that you can programmatically activate automatic restore, in order for WarmStandby to continue connection attempts in the background. The same client application may need to use both approaches, in order to start up by using the stop-on-pool-exhaustion approach (for example: a user authentication step on startup), and then activate automatic restore as soon as all the startup logic is done.

WarmStandby defines a concrete background automatic restore strategy that follows these rules:

- When an established connection to a server breaks (disconnection), the same server is retried immediately once (reconnection), in order to recover from casual network or protocol failures. A random delay can be configured (ReconnectionRandomDelayRange), which will be useful for client applications with a large number of running instances (such as custom agent desktops) so that every client does not try to reconnect at the same time.
- If reconnection fails, the next server in the pool is tried (failover). A configurable delay (BackupDelay) can be applied before the failover, for cases where a passive server may need some time to become active (such as a Configuration Server running in backup mode).
- After all the servers in the configured pool are tried, the pool is retried again, after a configurable delay (RetryDelay). This repeats indefinitely until the client application programmatically decides to stop. RetryDelay is a list, so that a back-off strategy can be applied to retry delays. For example: "first wait 5 seconds, then 10 seconds, and then 30 seconds for all future attempts."

Summary

The following is a brief summary of the different ways that client applications should use WarmStandby:

- Batch (synchronous, non-interactive) client applications will just need to call open(), and possibly check if the connection is open during their execution by using the isOpen() method.
- GUI (asynchronous, interactive) client applications will normally want to connect, but only keep the connection open after some other conditions hold (for example: user authentication, other connections also open, etc). They will therefore call openAsync() and then autoRestore() when appropriate.
- Daemon (lengthy, non-interactive) client applications can just call autoRestore() or, if they need to process the result of the open operation, the can use autoRestore(false) and then open() or openAsync().

Java

Creating

Before creating a new WarmStandby instance, you first create a protocol instance for the server you want to connect to. Every WarmStandby constructor requires a protocol instance as a parameter, as shown in the examples below.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs);
```

or

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
```

> **Important**
>
> Once the WarmStandby object is created, you can no longer use the open and close operations for that protocol or set the channel endpoint directly. These operations will now be handled using the WarmStandby object instead.

Configuring

The configuration for WarmStandby contains the following information:

- Endpoints: a list of endpoints which will be processed while trying to open the channel;
- Timeout: timeout for the channel opening operation;
- BackupDelay: interval between getting disconnected from a server and the first attempt to switch endpoints;
- RetryDelay: intervals between cycles for trying to reconnect to a server;
- ReconnectionRandomDelayRange: maximum value of additional random delay.

Initial configuration of WarmStandby occurs inside the instance constructor, but an external configuration can be applied whenever it is convenient for your application.

There are two ways you can update WarmStandby configuration:

1. directly updating specific configuration values in your WarmStandby instance
2. maintaining and updating a WSConfig object to hold configuration details, and then applying the entire configuration to your WarmStandby implementation

For simple applications where WarmStandby configuration typically does not change and you are connecting to a small number of Genesys servers, the first method may be easier. But if your application uses a more dynamic approach for the WarmStandby feature, or if you want to apply the same configuration details to multiple protocol objects, then using WSConfig to manage the configuration details can simplify your programming.

Updating Configuration Directly

You can use the getConfig() method to return and modify the current WarmStandby configuration details, as shown below.

```
ws.getConfig()
        .setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
        .setBackupDelay(2000)
        .setReconnectionRandomDelayRange(5000)
        .setRetryDelay(100, 500, 5000)
        .setTimeout(10000);
```

Note that you only need to set fields you want updated using this method. For example, if you use a constructor that sets Endpoint details then the setEndpoints line could be ignored.

Using the WSConfig Object

You can also create a WSConfig object that holds configuration details. This object allows you to update and manage configuration settings, and only have them applied to the WarmStandby object(s) when you are ready by using setConfig.

```
WSConfig  cfg = new WSConfig()
        .setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
        .setBackupDelay(2000)
        .setReconnectionRandomDelayRange(5000)
        .setRetryDelay(100, 500, 5000)
        .setTimeout(10000);

ws.setConfig(cfg);
```

or

```
WSConfig  cfg = new WSConfig();
cfg.setEndpoints(new Endpoint("host1", port1), new Endpoint("host2", port2))
cfg.setBackupDelay(2000)
cfg.setReconnectionRandomDelayRange(5000)
cfg.setRetryDelay(100, 500, 5000)
cfg.setTimeout(10000);

ws.setConfig(cfg);
```

Using WarmStandby

Opening a Protocol Without Reconnect

The following code shows how to make a single connection attempt. If this attempt is unsuccessful then WarmStandby finishes its work.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
try {
        ws.open();
}
catch (WSNoAvailableServersException ex) {
        // TODO: Handle exception
}
catch (WSCanceledException ex) {
        // TODO: Handle exception
}
```

or

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
try {
    ws.open();
}
catch (WSException ex) {
    // TODO: Handle exception
}
```

Opening a Protocol with Reconnect

The following code leads to an endless cycle of connection attempts, with some delays between attempts, until a success is found or a manual break occurs.

In this scenario, if a channel gets disconnected then WarmStandy will initiate a new cycle of connections to server.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
ws.autoRestore();
```

`WarmStandby.autoRestore()` is a way to instruct the WarmStandby: "you now take care of keeping the connection available in the background".

Closing a WarmStandby Connection

To close an open WarmStandby connection, use the close() method.

```
UniversalContactServerProtocol ucs = new UniversalContactServerProtocol();
WarmStandby ws = new WarmStandby(ucs, new Endpoint("host1", port1), new Endpoint("host2",
port2));
// TODO: do something
ws.close();
```

The close() method automatically cancels the requirement for any further attempts to re-establish a connection. Use the autoRestore(), or autoRestore(boolean), method to re-enable reconnection attempts.

Asynchronous Operations of WarmStandby

Code samples above are for synchronous operations, which block the current thread.

Asynchronous operations are similar to synchronous but don’t block the running thread. To use asynchronous operations, use the table below and replace any calls to synchronous methods with the asynchronous equivalents.

Synchronous method	Asynchronous equivalent
open()	openAsync().get()
close()	closeAsync().get()

> **Tip**
> You can also use the WarmStandby asynchronous open/close operations with the `Future` interface.

Using WarmStandby Event Handlers

WarmStandby contains four events you should use for notification of connection status.

Event name	Event description
ChannelOpened	Notifies that channel was opened successfully.
ChannelDisconnected	Notifies that channel was disconnected.
EndpointTriedUnsuccessfully	Notifies that the another connection attempt was unsuccessful.
AllEndpointsTriedUnsuccessfully	Notifies that the all connection attempts in the current cycle were unsuccessful.

> **Important**
> Using open() and openAsync().get() methods for your WarmStandby connection inside these event handlers will cause a thrown exception.

.NET

Creating

Before creating a new WarmStandby instance, you first create a protocol instance for the server you want to connect to. Every WarmStandby constructor requires a protocol instance as a parameter, as shown in the examples below.

```
var ws = new WarmStandby(new UniversalContactServerProtocol());
// any other child of ClientChannel may be used
```

> **Important**
> Once the WarmStandby object is created, you can no longer use the open and close operations for that protocol or set the channel endpoint directly. These operations will now be handled using the WarmStandby object instead.

Configuring

The configuration for WarmStandby contains the following information:

- Endpoints: a list of endpoints which will be processed while trying to open the channel;
- Timeout: timeout for the channel opening operation;
- BackupDelay: interval between getting disconnected from a server and the first attempt to switch endpoints;
- RetryDelay: intervals between cycles for trying to reconnect to a server;
- ReconnectionRandomDelayRange: maximum value of additional random delay.

Initial configuration of WarmStandby occurs inside the instance constructor, but you can also use the WSConfig object to hold a custom values that can be maintained and applied whenever it is convenient for your application. WSConfig allows your application to adjust the WarmStandby configuration details as needed, or allows you to apply the same configuration details to multiple protocol objects.

There are two ways to create a WSConfig object:

```
var cfg = new WSConfig
{
      Endpoints = new List<Endpoint>
      {
            new Endpoint("host1", port1),
            new Endpoint("host2", port2),
            new Endpoint("host3", port3)
      },
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 5000,
      RetryDelay = new []{100, 500, 5000},
      Timeout = 10000
};
```

or

```
cfg = new WSConfig();
cfg.SetEndpoints(new Endpoint("host1", port1),
      new Endpoint("host2", port2),
      new Endpoint("host3", port3));
cfg.SetRetryDelay(100, 500, 5000);
cfg.BackupDelay = 2000;
cfg.Timeout = 10000;
cfg.ReconnectionRandomDelayRange = 5000;
```

Then you can easily apply the configuration to an existing WarmStandby instance:

```
ws.Configuration = cfg;
```

Using WarmStandby

Opening a Protocol Without Reconnect

The following code shows how to make a single connection attempt. If this attempt is unsuccessful then WarmStandby finishes its work.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
var cfg = new WSConfig()
{
      Timeout = 1000,
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 3000
}.SetEndpoints(new Endpoint("host1", port1), new Endpoint("host2",
port2)).SetRetryDelay(1000, 2000);
ws.Configuration = cfg;
try
{
      ws.Open();
}
catch (Exception)
{
      // TODO: Handle exception
}
```

Opening a Protocol with Reconnect

The following code leads to an endless cycle of connection attempts, with some delays between attempts, until a success is found or a manual break occurs.

In this scenario, if a channel gets disconnected then WarmStandy will initiate a new cycle of connections to server.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
var cfg = new WSConfig()
{
      Timeout = 1000,
      BackupDelay = 2000,
      ReconnectionRandomDelayRange = 3000
}.SetEndpoints(new Endpoint("host1", port1), new Endpoint("host2",
port2)).SetRetryDelay(1000, 2000);
ws.Configuration = cfg;
ws.AutoRestore(true); // leads to opening of Warmstandby
```

Closing a WarmStandby Connection

To close an open WarmStandby connection, use the Close() method.

```
var ws = new WarmStandby(new UniversalContactServerProtocol ());
// any other child of ClientChannel may be used
// TODO: do something
try
{
      ws.Close();
```

```
}
catch (Exception)
{
      // TODO: Handle exception
}
```

The Close() method automatically cancels the requirement for any further attempts to re-establish a connection. Use the AutoRestore(), or AutoRestore(bool), method to re-enable reconnection attempts.

Asynchronous Operations of WarmStandby

Code samples above are for synchronous operations, which lock the current thread.

Asynchronous operations are similar to synchronous but don't lock the running thread. To use asynchronous operations, use the table below and replace any calls to synchronous methods with the asynchronous equivalents.

Synchronous method	Asynchronous equivalent
Open()	EndOpen(BeginOpen(null,null))
Close()	EndClose(BeginClose(null,null))

> **Tip**
>
> WarmStandby .NET asynchronous open/close operations were designed using `IAsyncResult` interface.

Using WarmStandby Event Handlers

WarmStandby contains four events you should use for notification of connection status.

Event name	Event description
ChannelOpened	Notifies that channel was opened successfully.
ChannelDisconnected	Notifies that channel was disconnected.
EndpointTriedUnsuccessfully	Notifies that the another connection attempt was unsuccessful.
AllEndpointsTriedUnsuccessfully	Notifies that the all connection attempts in the current cycle were unsuccessful.

> **Important**
>
> Using the Open() or EndOpen(IAsyncResult) methods for your WarmStandby connection inside these event handlers will cause a thrown exception.