Continuous build and deploy automation with IBM® Integration Bus V10 using Ant, Git, and Jenkins

Geza Geleji
Published on October 2, 2015 / Updated on November 22, 2017

The build process in IBM® Integration Toolkit Version 10 Fix Pack 2 lends itself to straightforward automation using various open source components. In this post, we demonstrate a means of achieving continuous integration using some popular open source technologies, such as Git (version 1.7.1) for version control, Apache Ant (version 1.7.1) for build automation, and Jenkins (version 1.630) for continuous integration.

A number of articles have been published on automating the installation of the IBM Integration Bus runtime using software such as IBM UrbanCode Deploy and Chef; for a summary of these, please refer to How to automate IBM Integration Bus deployments using IBM UrbanCode Deploy and Chef by Simon Holdsworth. His article describes, among other things, a way to deploy pre-packaged integration solutions (BAR files) to the IBM Integration Bus runtime. While we also give a possible solution to the same problem, we will be more focused on building and packaging the integration applications efficiently.

Preliminaries

The build environment used in this demonstration consists of Git, Apache Ant and Jenkins along with IBM Integration Bus Version 10 Fix Pack 2 running on Red Hat Enterprise Linux Server release 6.6. We also make use of a development environment on the same host, running the IBM Integration Toolkit, and hosting a Git repository which acts as, using Git terminology, the remote origin for the build process.

We are looking to automate the following segment of a typical development & test workflow (Figure 1):

Figure 1: automated build & deployment workflow

1. Someone pushes an update to a Git repository which is configured to notify Jenkins of the update.
2. Jenkins pulls the update into its own clone of the Git repository.
3. The build process compiles all IIB source artefacts.
4. Results are packaged in a BAR file.
5. The packaged artefacts are configured for deployment.
6. The BAR file is deployed to a test integration server.
7. The changes from the Git update in step 1 take effect on the integration server.
8. The newly deployed artefacts are available for testing.

The most obvious advantage here is that a Git commit may automatically get published to the test integration server, either immediately, or at pre-defined intervals. The build process is managed entirely by Jenkins. Once the build environment is in place, all we need to do is configure Git to notify Jenkins when a push occurs to the repository.

Implementation

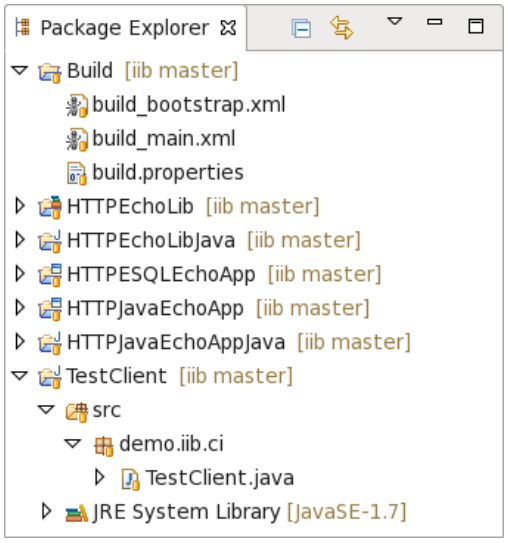

Figure 2: source artefacts

Our sample implementation makes use of a pair of Ant scripts that perform a build on the Integration Toolkit workspace, which is automatically cloned by Jenkins to a temporary location before the build scripts are called. Jenkins is configured to invoke the first build script, which we call `build_bootstrap.xml`; this, in turn, calls the other script (`build_main.xml`) in a new shell running under a different user ID, with the IBM Integration Bus shell environment sourced to allow easy execution of IIB administrative commands under appropriate user privileges. We implement this privilege shift with the Unix `sudo` command.

Some parts of these scripts depend heavily on the operating system they are being run on. The examples we provide have been tested on Red Hat Enterprise Linux 6.6, and modifications may be necessary for different OS versions or distributions.

This demonstration contains two Integration Applications that will be built and deployed by Jenkins: `HTTPESQLEchoApp` and `HTTPJavaEchoApp`. Both applications accept short text input messages over the HTTP protocol, do a trivial transformation, and send the result back as a reply over the connection.

`HTTPESQLEchoApp` (see Figure 3.a) does the transformation using an ESQL compute node. ESQL code can be directly deployed to the integration runtime in IBM Integration Bus V10, and no special build steps need to be taken.

`HTTPJavaEchoApp` (see Figure 3.b), on the other hand, includes Java code which needs to be compiled to bytecode. This compilation step must be performed either before or at some point during the build process.

For the sake of the demonstration, both applications contain their respective transformation code in a shared library called `HTTPEchoLib`. This is referenced from both applications, and is handled appropriately during the build.

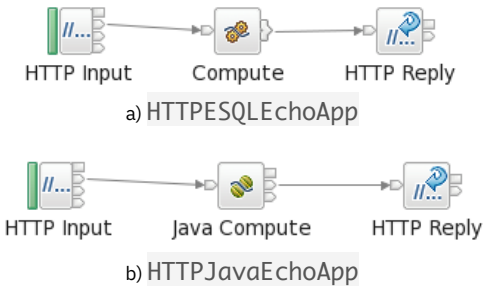

Figure 3: test message flows

We also include a Java test application called `TestClient` to verify our deployment. This is automatically built with the rest of the materials and invoked at the end of the build process. It sends a short test message to both `HTTPESQLEchoApp` and `HTTPJavaEchoApp`, and prints the response to the console, to appear in the Jenkins build logs. These log entries will help us ensure that the deployed applications are working as expected.

The contents of the IBM Integration Toolkit workspace used herein is attached as `iib-ci-repo.zip`. This archive contains all the source materials needed for running this demonstration.

Preparing the environment

For the sake of completeness, we will give instructions on setting up a build environment, assuming a clean install of Red Hat Enterprise Linux Server 6.6, with an X Window System graphical interface, and an SSH server readily available.

1) First of all, the Ant and Git software packages need to be installed. On RHEL 6.6, the following command installs Ant 1.7.1 and Git 1.7.1:

```
yum install ant ant-nodeps git
```

If we don't already have the X Virtual Framebuffer installed, we should also do

```
yum install xorg-x11-server-Xvfb
```

Note that Xvfb is an optional package on RHEL that may need to be downloaded separately.

2) We then need to install the latest fix pack level of IBM Integration Bus Version 10 (Fix pack 2 at the time this post was published). IBM MQ is not required for this demo.

3) Create a user which runs IBM Integration Bus and hosts the Git repository:

```
useradd -m demo
usermod -a -G mqbrkrs demo
```

4) Download, then install Jenkins:

```
rpm -i jenkins-1.630-1.1.noarch.rpm
```

Now run `visudo` as `root`, and make sure the following lines are present in the `/etc/sudoers` file in the order given below (though there may be other lines between them):

```
Defaults requiretty
Defaults:%wheel !requiretty
%wheel ALL=(ALL) NOPASSWD: ALL
```

Add user `jenkins` to the `wheel` group:

```
usermod -a -G wheel jenkins
```

The above is to allow Jenkins to make use of the `sudo` command from a build script. We can now start Jenkins:

```
service jenkins start
```

We may also need to configure security, and update already installed plugins to their latest versions. We will also need Jenkins' Git plugin and the Xvfb plugin. Both can be installed from within the Jenkins console.

5) Create a new Git repository on the build server in, e.g., `/home/demo/git/iib.git`:

```
mkdir -p /home/demo/git/iib.git
cd /home/demo/git/iib.git
git init --bare
```

The workspace contained in the `iib-ci-repo.zip` archive attached to this post can be imported using

```
mkdir /home/demo/git/tmp
cd /home/demo/git/tmp
git clone file:///home/demo/git/iib/
cd iib
unzip <path_to>/iib-ci-repo.zip
git add *
git commit -m "Initial import"
git push --all
```

Create a Git post-receive hook to automatically notify Jenkins of changes to the Git repository:

```
echo "curl http://localhost:8080/git/notifyCommit?url=ssh://localhost:22/home/demo/git/iib.gi
  t/" > /home/demo/git/iib.git/hooks/post-receive
chmod +x /home/demo/git/iib.git/hooks/post-receive
```

6) Create an SSH keypair for user `demo` to allow Jenkins SSH access to the Git repository:

```
ssh-keygen -t rsa -b 4096 -C "demo"
```

The public key needs to be installed in the `authorized_keys` file on the SSH server, whereas the private key should be stored in Jenkins. From the top level of the Jenkins console, navigate to 'Credentials', 'Global credentials (unrestricted)', and select 'Add Credentials'. Choose 'SSH Username with private key' in the 'Kind' drop-down box, enter 'demo' as the username, and import the private key as appropriate.

7) Define an Xvfb installation in Jenkins, under 'Manage Jenkins' / 'Configure System' / 'Xvfb installation'; enter Xvfb1 as name, and `/usr/bin` for the location of the Xvfb executable (this is the default under RHEL 6.6).

8) Create a new job in Jenkins by navigating to the top level of the console,

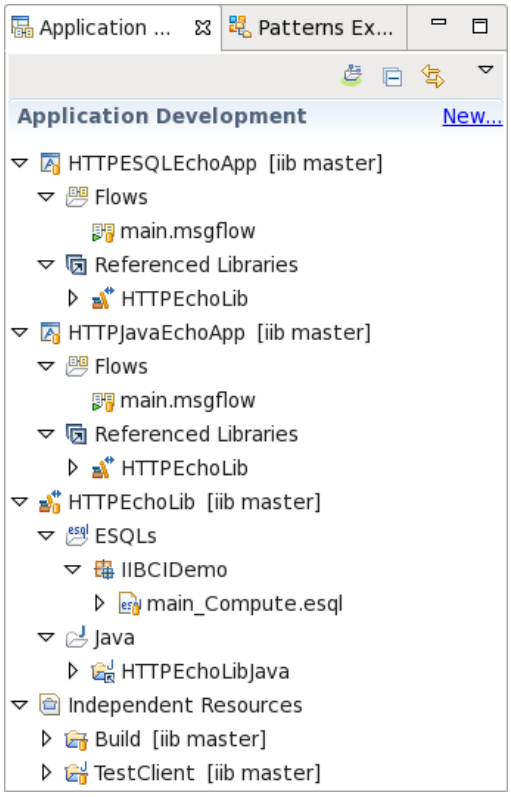

Figure 4: source artefacts in the Integration Toolkit

selecting 'New Item' from the left-hand menu, choosing a name such as 'IIB-CI' and the option 'Freestyle project', then clicking OK. On the configuration panel that follows:
Select 'Git' under 'Source Code Management',
specify `ssh://demo@localhost/home/demo/git/iib.git/` as 'Repository URL' and select 'demo' from 'Credentials'. Jenkins will immediately attempt to connect, and you should not see any errors.

Set 'Poll SCM' under 'Build Triggers', and set 'Schedule' to 'H H 1 1 *'.

Select 'Start Xvfb before the build, and shut it down after.' under 'Build Environment'; click the 'Advanced...' button, set 'Xvfb installation' to the installation created earlier ('Xvfb1', and 'Xvfb display name offset' to an appropriate value (e.g., 100) to be used as an X Window System display that is not used for any other purpose on the system.

Under 'Build', click 'Add build step', select 'Invoke Ant' from the pop-down menu; click 'Advanced...', then set 'Targets' to `-e` (to prevent Ant from adorning each line in the log excessively with prefixes), and 'Build File' to `Build/build_bootstrap.xml`.

Save the project.

9) Create integration node `NODE1` and server `svr1` on the build host as user `demo`:

```
. /opt/mqsi/iib-10.0.0.2/server/bin/mqsiprofile
mqsicreatebroker NODE1
mqsistart NODE1
mqsicreateexecutiongroup NODE1 -e svr1
```

10) Start the Integration Toolkit, and create a new workspace.
Install "Eclipse Git Team Provider" from the `http://download.eclipse.org/egit/updates/` Eclipse update site.

Switch to the Git perspective, and select "Clone a Git repository".

Set the URI to `ssh://demo@localhost:22/home/demo/git/iib.git/`. Ensure that "master" is ticked on the panel that follows.

Tick "Import all existing Eclipse projects after clone finishes" on the final panel. The resulting project layout should resemble those seen on Figures 2 and 4.

At this point, the build environment should be fully functional. Review the `Build/build.properties` file to make sure the settings match the environment, with special regard to the IBM Integration Bus installation directory (`path.iib`). Test the process by making a change to any of the project files (e.g., move a node in a message flow on the canvas, then save the message flow), then commit and push to Git. The build process should be automatically triggered, and successful execution is indicated by the presence of the following excerpt at the end of the build log as seen in the Jenkins console output:

```
    [exec] BIP8071I: Successful command completion.
    [[java]] ------------------------------
    [[java]] path.iib="/opt/mqsi/iib-10.0.0.2"
    [[java]] http.esql.echo.url="HTTPESQLTest"
    [[java]] iib.bar="test.bar"
    [[java]] ant.main.class="org.apache.tools.ant.Main"
    [[java]] iib.node="NODE1"
```

```
    [[java]] user.iib="demo"
    [[java]] http.java.echo.url="HTTPJavaTest"
    [[java]] iib.svr="svr1"
    [[java]] iib.mqsiprofile="${path.iib}/server/bin/mqsiprofile"
    [[java]] group.iib="mqbrkrs"
    [[java]] ant.launcher.class="org.apache.tools.ant.launch.Launcher"
    [[java]] http.port="7800"
    [[java]] -------- IIB HTTP test --------
    [[java]] http://localhost:7800/HTTPESQLTest
    [[java]] HTTP response: 200 OK
    [[java]] "HTTPESQLTest: foo TestRequestMessage bar"
    [[java]] -------- IIB HTTP test --------
    [[java]] http://localhost:7800/HTTPJavaTest
    [[java]] HTTP response: 200 OK
    [[java]] "HTTPJavaTest: foo TestRequestMessage bar"

BUILD SUCCESSFUL
Total time: 1 minute 7 seconds

BUILD SUCCESSFUL
Total time: 1 minute 8 seconds
Xvfb stopping
Finished: SUCCESS
```

The presence of the above shows that the Toolkit workspace has been built successfully, a BAR file has been produced and deployed to the Integration Server, and that the `TestClient` application was able to send a test message through HTTP and get the expected response back.

How does it work?

As mentioned earlier, our build process relies on Ant scripts to perform the necessary work, even though using Ant is not the only option here. Jenkins is highly configurable and extensible through readily available plugins, and integrates well with various open source and proprietary software production tools. It provides a stable environment for automatically running the build process when needed, allowing developers to focus more on application development and less on trivial production tasks.

The `build_bootstrap.xml` script sets up the Integration Bus administrative environment, and calls the `build_main.xml` script under a different user inside it. This serves to isolate the Jenkins build environment from the IIB administrative environment. The access permission of the workspace files cloned from Git are also updated.

Given that it is running under IIB administrative privileges, the `build_main.xml` script may perform workspace build tasks as well as IIB administrative tasks such as deployment and configuration. The first and most important step is to produce properly configured, deployable artefacts from the workspace source materials. There are two basic methods of achieving this:

- The `mqsicreatebar` command
- The `mqsipackagebar` command

The main difference is that `mqsicreatebar` is a headless Integration Toolkit command capable of performing a full build of a Toolkit workspace, including Java and message set compilation; `mqsipackagebar`, on the other hand, only adds deployable resources to a BAR file, and therefore may not always be applicable. The latter command can be used to create BAR files on computers that do not have IBM Integration Toolkit installed. Java source code and message sets either need to be compiled separately, or you need to use the `mqsicreatebar` command to compile and/or package them. For further information, refer to section Packaging resources that include Java code or message sets in the IBM Integration Bus V10 Knowledge Center.

The way our example invokes the `mqsicreatebar` command is

```
<exec executable="mqsicreatebar">
  <arg value="-data" />
  <arg value="${basedir}" />
  <arg value="-b" />
  <arg value="${iib.bar}" />
  <arg value="-p" />
  <arg value="TestClient" />
  <arg value="-a" />
  <arg value="HTTPESQLEchoApp" />
  <arg value="HTTPJavaEchoApp" />
  <arg value="-l" />
  <arg value="HTTPEchoLib" />
  <arg value="-deployAsSource" />
```

```
</exec>
```

`-data` and `${basedir}` declare the Toolkit workspace to be built. `${basedir}` is a property supplied by Ant that one may set on the `project` tag of the Ant build script. Since it is resolved relative to the build script file, we can easily use it to identify the root of the workspace.

`-b` and `${iib.bar}` define the output BAR file to be built. Specified relative to the current working directory, this will be a file in the workspace root, as declared by the `iib.bar` property in the `build.properties` file.

The `-p` argument declares `TestClient` as a legacy Eclipse project to be built. In our case, this is a Java application and it will get compiled as such.

`-a` designates `HTTPESQLEchoApp` and `HTTPJavaEchoApp` as IIB applications, and `-l` specifies that `HTTPEchoLib` is an IIB library (in fact, it is referenced by the two applications).

`-deployAsSource` prevents message flows from being compiled, and ESQL resources from being inlined. We are building a shared library (`HTTPEchoLib`) in this step, and resources in shared libraries cannot be inlined.

Instead of `mqsicreatebar`, we could also use `mqsipackagebar`, which would be invoked as follows:

```
<exec executable="mqsipackagebar">
  <arg value="-a" />
  <arg value="${iib.bar}" />
  <arg value="-k" />
  <arg value="HTTPESQLEchoApp" />
  <arg value="HTTPJavaEchoApp" />
  <arg value="-y" />
  <arg value="HTTPEchoLib" />
</exec>
```

We do not specify a workspace directory here (this would be done using the `-w` argument), so the command would assume that the current working directory is the workspace root.

`-a` declares the output BAR file to be created, `-k` designates our two applications, and `-y` our library.

Since this command is not capable of compiling Java source files, these would already have to be present in the workspace before the command is invoked. We could either store the Java class files in the Git repository, or compile Java sources separately, e.g., using the Ant `javac` task.

At this point, we have an almost-deployable BAR file, on which certain configuration steps still need to be performed. As the source artefacts coming from Git may need to be run in various different environments, they can not be expected to have all settings, such as endpoints, names, timeouts, etc. readily configured. The `mqsiapplybaroverride` command can help us in gaining this flexibility, as it allows a wide range of properties to be set on nodes and message flows after the BAR file has been built.

To take an example from our build script, let us see how an HTTP endpoint URL is configured:

```
<exec executable="mqsiapplybaroverride">
  <arg value="-b" />
  <arg value="${iib.bar}" />
  <arg value="-m" />
  <arg value="main#HTTP Input.URLSpecifier=/${http.esql.echo.url}" />
  <arg value="-k" />
  <arg value="HTTPESQLEchoApp" />
</exec>
```

Here, `-b` designates the BAR file, and `-k` the application that we would like to configure. `-m` is followed by a `key=value` pair setting a configuration parameter — in our case, the `URLSpecifier` property of the `HTTP Input` node of the `main` message flow, to a value derived from the `http.esql.echo.url` Ant property declared in the `build.properties` file.

After updating the configuration parameters, we are ready to deploy the BAR file to our integration server using the `mqsideploy` command. We then call `mqsireportproperties` to ensure that the deployment has taken effect before starting our test client to put a message through and verify the functional behaviour of the two applications that have been deployed.

Putting it into production

The above demonstration is a vastly simplified analogy of a proper development and test system. All software components involved are capable of handling much more sophisticated scenarios.

For example, in a production development / test system, the Git repository would probably not be local to the build server, and therefore the Git SSH URLs would point to remote hosts. However, the use of the SSH protocol for Git access will easily allow this change to be made. Builds could be triggered by changes originating from any of a large number of sources, and multiple build servers would be handling the workload.

Stricter security controls would be in place, primarily relying on the access control facilities of Jenkins and Git. Parts of the build process is performed under the privileges of the Jenkins user, others under the IIB administrative user. The boundary where the handover occurs needs to be set appropriately for a more complex environment.

Configuration of the deployable artefacts will likely depend on the specific environment in which the build and/or test is occurring, with lots of configuration parameters possibly derived from volatile environmental properties. The build and test runtime environments could be separated even further: `mqsideploy` may be configured to deploy the build output to a remote Integration Server, where much more complex test procedures can be executed.

Conclusion

The above is a simplified illustration of a develop–deploy–test cycle as it could be implemented with IBM Integration Bus, Jenkins, Git, and Ant. We have demonstrated use cases of the build commands provided by IIB, and have seen a way to package development artefacts into a self-contained, portable archive suitable for later deployment.

Readers are encouraged to follow and reconstruct this process themselves. We have made all source materials used in this post available in the archive attached as `iib-ci-repo.zip`, and have strived to make the instructions as complete as possible. They have been verified to work in the stated environment, however, the presence of errors is always a possibility. We recommend that the entire process be carefully verified, and improvements be made as appropriate.

Acknowledgements

The author would like to thank Tim Dunn, David Gorman, Martin Ross, and Karen Cameron for their helpful suggestions.

TAGS IBM-INTEGRATION-BUS, INTEGRATION BUS, INTEGRATION-TOOLKIT, CONTINUOUS-INTEGRATION, ANT, GIT, JENKINS

Geza Geleji

---

27 comments on"Continuous build and deploy automation with IBM® Integration Bus V10 using Ant, Git, and Jenkins"

Sunny Ghanathey February 24, 2020

Hi,

My build says Successfully deployed the bar files to “server”,but when i go and check my server and i dont see its deployed.
can you guys tell ,what would be reason?

Thanks

Reply (Edit)

Mangesh Kumbhar March 27, 2019

Really nice and helpful article.

Reply (Edit)

Rajeshwar Tiwari September 26, 2017

Is there any way to create bar using Maven ?

Reply (Edit)

Kumar September 05, 2017

Hi,

Im trying implement CI/CD in windows. I'm unable to run mqsiprofile command using ant. I have used same PI which has been used in this post with minimal changes done to support windows like RUNAS,ICACLS to get the user access.

Reply (Edit)

> Micha August 08, 2018
>
> is the best solution without calling mqsiprofile in a batch file.
>
> Reply (Edit)

Name *Aroni April 27, 2017

Hi. Geza. Could you inform what is the egit plugin version (windows 10) for IIB v10?

Reply (Edit)

> kurojiyu June 14, 2017
>
> http://wiki.eclipse.org/EGit/FAQ#Where_can_I_find_older_releases_of_EGit.3F
>
> here you find all versions, for IIB v10 i works with 2.3.1.201302201838-r version.
>
> Reply (Edit)

Harry January 03, 2017

Hello,
Very intersting blog. Can you also please write about this same CI of IIB in Windows platform...

Reply (Edit)

> Sandeep104 July 24, 2017
>
> Great stuff...can we have similar article for Windows instead of Linux?
>
> Reply (Edit)

> Steven Ricks August 14, 2017
>
> I too would like a Windows version please.....
>
> Reply (Edit)

> > Geza Geleji August 14, 2017
> >
> > I'm afraid you would need to get Git over SSH working first to get the exact same thing done on Windows. If you only need to run the Git client side on Windows, that should probably be fairly straightforward.
> >
> > Reply (Edit)

Long July 25, 2016

Hello,
I get the error shown below in the Jenkins' Console Output (running in Firefox) despite following your instructions to the details. In my case it may have to do with the versions of the software I am using (Red Hat 7.2, IIBv10.0.0.5, Jenkins 2.15-1.1).

Do you have any suggestion of how to circumvent the error?

“[exec] BIP0965E Error compiling files in mqsicreatebar.
[exec]
[exec] The message is:
[exec] A required display could not be obtained from the underlying operating system. To resolve this issue on Linux install X Virtual Frame Buffer (Xvfb), start Xvfb, and set the DISPLAY environment variable on your system.
[exec]
[exec]
[exec] Result: 1
[exec] BIP1049E: The BAR file does not exist.”

Reply (Edit)

Geza Geleji July 26, 2016

Hi Long,
it seems that something may be amiss in your Xvfb configuration. Perhaps the Xvfb plugin in Jenkins is not configured properly? I tried this on RHEL 7.2 and had no problems.

Reply (Edit)

Long July 26, 2016

Thank you Geza for your quick comment. I will try it again with an older version of Jenkins and will let you know the outcome. BTW, what version of Jenkins you’re using in Red Hat 7.2?

Reply (Edit)

Long July 26, 2016

I forgot to add that the new version of Jenkins would have the Xvfb configuration in ‘Manage Jenkins->Global Tool Configuration’ instead of in ‘Manage Jenkins’->‘Configure System’. I don’t know if that would make a difference? In the newer of Jenkins, I also have to install the Xvfb plugin as opposed to it comes with Jenkins as default.

Reply (Edit)

Long July 27, 2016

Hello Geza,
I successfully have it working. The step I missed out previously was to enable Xvfg in the project!

Here’s the sequence of steps:
1. Install jenkins
2. install Xvfb plugin
3. Configure Xvfb (in Manage Jenkins->Global Tool Configuration instead of Manage Jenkins->Configure System)
4. In a jenkins project (e.g. New Item) under Build Environment tab, check to enable the ‘Start Xvfb before the build, and it down after’ option. The default is not check/enable and hence I encountered the error.

I appreciate you have provided us with such an excellent guide. Thank you.

Reply (Edit)

Nihar February 23, 2016

Hi Geza,
This is a nice article.
I am trying to achieve the similar goals for Continuous build and deploy automation with IBM Integration Bus V9 using Rational Build Forge 7.1.3.3. Is this something you have had a loook at or can help me with?
Thanks & Regards,
Nihar.

Reply (Edit)

Geza Geleji February 23, 2016

Hi Nihar, I am sorry, but I am not familiar with Build Forge.

Reply (Edit)

Sujeeth November 27, 2015

Hey Geza,

Indeed, knowledge center “assumes” linux for desktop version (GUI mode) but it is not clear anywhere how to use toolkit specific commands in linux server variants (e.g. CLI).

Adding such information here will bring up some clue / value to this post, I think..

Cheers !

Reply (Edit)

Geza Geleji November 30, 2015

In a purely command-line environment, you need to use Xvfb to provide an X11 server for mqsicreatebar (see https://developer.ibm.com/answers/questions/233611/bip0965e-error-compiling-files-in-mqsicreatebar.html). Note that mqsipackagebar doesn’t have this requirement, so it may be better suited for the purpose, even though it provides slightly different functionality.

Reply (Edit)

Sujeeth Pakala December 03, 2015

Got it working after installing Xvfb and its relevant plugin in jenkins.

But, were you sourcing mqsiprofile for each build from ant? Is there a better way ?

I sourced it in the jenkins (service user) profile. But, it is not used during build from jenkins.

Reply (Edit)

Geza Geleji December 03, 2015

Yes, I am sourcing mqsiprofile for each build. This has to be done from the shell instance in which Jenkins is running the build process. The overhead is very minimal, and sourcing it every time allows any changes made to the script (e.g., by a product update) get picked automatically on the next run without the need for manual intervention.

Reply (Edit)

Sujeeth Pakala December 03, 2015

Okay !

Jenkins sources its profile into /etc/sysconfig/jenkins.

I have used mqsiprofile command there. In case of product update, just restart jenkins, you will have latest profile sourced.

(Edit)

bill whiting March 27, 2017

Is the need for Xvfb new to v10. That wasn’t required in v7 or v9.
Is that a bug introduced by a newer version of eclipse?

Reply (Edit)

Geza Geleji April 04, 2017

Yes, I believe this was introduced by an Eclipse upgrade. More details with a link to the relevant Eclipse work item are available at org.eclipse.swt.SWTError: No more handles [gtk_init_check() failed] error when running headless build on Linux using IBM Integration Designer

Reply (Edit)

Sujeeth November 27, 2015

Informative. But it is not clear how to run mqsicreatebar command in red hat linux (command line version).

Reply (Edit)

Geza Geleji November 27, 2015

Hi Sujeeth,
this is just an example. A complete description of using the mqsicreatebar command can be found in the Knowledge Center at mqsicreatebar command.

Reply (Edit)