IBM z/OS application continuous integration: Part 1.
Building continuous integration with Rational Team Concert
Zhang Hong Chen
Rosalind Radcliffe
November 13, 2012

Continuous integration has become an important practice of agile software development, because it improves software quality by continuously applying a quality control process. This article describes practices of mapping continuous integration concepts that are used in distributed development to the IBM System z domain. You'll learn how to use Rational Team Concert to automate the build and deployment of a mainframe application. The second article of this two-part series describes how to add automated unit tests and build verification tests to the continuous integration process.

Continuous integration overview

When new development tasks are assigned to them, developers get copies of the code from the code repository to work on. The longer the developers work separately on their local copies of the code, the greater the risk for difficulties when they try to integrate their work in the integration stream. In a continuous integration approach, developers deliver code changes frequently to get feedback from automated build and test processes, thus reducing the risk and effort of late integration.

These are among best practices to follow when adopting continuous integration:

- Maintain a code repository
- Automate builds
- Automate deployment
- Make results of the latest build available to everyone on the team
- Automate unit tests
- Automate function tests

In the following sections, we demonstrate the use of IBM® Rational Team Concert™ in the first four of those practices for IBM® z/OS® application development. Each of the practices is then tied together to create a continuous integration build.
We cover the automated testing topics in the second article in this two-part series.

**Maintain a code repository**

For source control, it is important to understand that Rational Team Concert uses a stream, components, a workspace, and change sets to manage source code. The source files are stored in the repository (the database) as the definitive source and are extracted to a file system for editing or building. You can work locally with your files, deliver changes to a stream for version control, and then have a build agent load them onto z/OS for building. See the Resources section for a link to the article titled “Guide for migration to Rational Team Concert for z/OS application development” for a more detailed description.

It is a best practice to create a dedicated repository workspace for a team build. Dedicated build workspaces enable you to identify changes and work items that go into a specific build and to create snapshots of the build source code. They also enable you to build only when you have changes in the stream. Figure 1 shows a common workflow.

![Figure 1. Workflow from individual developers to the build workspace](image)

Developers do unit test by using the personal build capability and the Rational Development and Test Environment for System z (see Resources for more information). When code is unit tested and delivered to the development steam, it is automatically accepted by the build workspace and loaded to a team build environment to do the integration build.

**Automate builds**

Rational Team Concert provides many capabilities for calling existing build tools and provides a feature-rich, dependency-based build. All of these types of builds are included in build definitions and can be configured to run on a schedule. Figure 2 shows a build that is scheduled to run every 60 minutes.
Some practices recommend that you rebuild whenever new code is delivered. This depends on another practice to keep the build fast. Rational Team Concert does not automatically perform a build on delivery of code. You can achieve the goal of building early for a change by scheduling your build with a short interval and enabling the option to **Build only if there are changes accepted**. This will make Rational Team Concert check your team's stream frequently and rebuild whenever a change has been delivered to it.

**Tip:**
When you choose this option, it is important to have a dedicated build workspace to avoid missing build execution because another build has accepted the changes. Figure 3 shows this option in the Build Definition view.
Figure 3. Automatic build on delivery of code but only if changes are accepted

Automate deployment

To do more than simple validation (and even for simple validation with many applications), the application needs to be deployed to a target test environment. The next sections explain the multiple considerations for the deployment of an application for testing.

Move application executable files

The application executables that are the results of build need to be deployed to the test environment. In Rational Team Concert, you use the Package function to gather your build artifacts and create a package that can then be deployed to another system or a different location in the same system.
Install the application and set up the infrastructure

Some applications need an installation step to be ready for testing (for example, the CICS NEWCOPY command to load the new version of the program). When you add a new module to the application, it might also need new infrastructure setup before the new module can run. For example, when you add a new screen to a CICS application, it is likely that you will need a new MAPSET definition.
This is done with TSO commands or scripts, such as REXX or CLIST, and submitted through JCL (job control language). These scripts can be run automatically during deployment at the appropriate stage.

Figure 6 shows a deployment definition that executes the NEWCOPY command, a REXX script, after the files have been deployed to the defined PDS.

**Figure 6. Post-deployment command**

![Deployment Definition](image)

It is a best practice to manage your infrastructure setup and configuration code in your code repository, just as what you do for your application source code. Infrastructure code can be packaged and deployed with your application source code, and that makes automation easier and less error-prone. The concept goes along with the *Infrastructure as Code practices* when people are talking about DevOps. By managing infrastructure as code in a code repository, you get all of the benefit that you normally get with source control, such as the latest central version of the script.

**Install the test data**

*Managing test data* is a big topic. The question about continuous integration is usually how to get the test data automatically set up or restored in the test environment. The approach we used for deploying application executables and setting up the infrastructure can be used to manage test data, as well.

Data management tools such as the IDCAMS utility for Virtual Storage Access Method (VSAM) or enterprise tools such as IBM® Optim™ Test Data Management (see Resources) can export your test data to a file system and restore it to a database in your test environment.
The exported data, in the form of a data set, can be packaged and deployed to your test environment. Deploy post-command can be used to call the batch functions of the data management tool to restore the data file to the test database.

Depending on whether you want your test data be reinstalled every time your application tests run, you can choose to reuse the application package and deployment definition or create a new set of test data package and deployment definitions.

Figure 7 shows how IBM Optim Test Data Management can be used in the continuous integration context. Integration with Rational Team Concert makes test data deployment part of the automated integration cycle.

**Figure 7. Optim Test Data Management in continuous integration**

Create a continuous integration build

You can build, package, deploy, set up infrastructure, and test data installation as separate tasks. Now it is time to connect everything and create a continuous integration (CI) build.

Rational Team Concert provides a set of Apache Ant tasks to control the build lifecycle. Package and Deploy are technically a special type of Rational Team Concert build, so they can be managed by using the ANT task, as well. See Resources for a link to the requestTeamBuild section of the documentation for a complete reference of the Ant tasks.

Create the CI build as a Rational Team Concert build, and select the Ant with Enterprise Extensions – Rational Build Agent template (see Figure 8). That template is a build definition for
z/OS that uses the Rational Build Agent build engine and a version of the Ant build toolkit that has been extended to work on z/OS.

**Figure 8. Available build templates**

![Available build templates](image)

In the CI build definition (Figure 9), choose an existing build file, and point to the location of the Ant script. You can either manually place the Ant script in the target system or have Rational Team Concert load it from a source project before the build starts. However, it is best to store your build script in a project in your code repository and have Rational Team Concert load it before the build. You get the same benefits as described in managing infrastructure setup code.

**Figure 9. Specify an existing build file**

![Specify an existing build file](image)
The code sample in Listing 1 shows how to use Rational Team Concert Ant tasks to perform these tasks:

- Request a mortgage.dev build by using the `requestTeamBuild` task.
- Wait for a build to finish by using the `waitForTeamBuild` task.
- Report build activity status to the CI build result by using `startBuildActivity` task.
- Publish result links to the CI build result by using `linkPublisher` task.
- Check the status of the mortgage.dev build and control the process of a CI build.

Listing 1.

```xml
<target description="step1" name="step1">
  <startBuildActivity autoComplete="true" buildResultUUID="${buildResultUUID}" label="Enterprise build" parentActivityId="${teamz.build.ant.buildActivityId}" passwordFile="${passwordFile}" repositoryAddress="${repositoryAddress}" userId="${userId}" />
  
  <requestTeamBuild buildDefinitionId="mortgage.dev" passwordFile="${passwordFile}" repositoryAddress="${repositoryAddress}" userId="${userId}" resultUUIDProperty="step1BuildResultUUID" />

  <linkPublisher repositoryAddress="${repositoryAddress}" userId="${userId}" passwordFile="${passwordFile}" buildResultUUID="${buildResultUUID}" url="${repositoryAddress}/resource/itemOid/com.ibm.team.build.BuildResult/${step1BuildResultUUID}" label="mortgage.dev build result" />

  <waitForTeamBuild repositoryAddress="${repositoryAddress}" passwordFile="${passwordFile}" userId="${userId}" buildResultUUID="${step1BuildResultUUID}"statesToWaitFor="COMPLETED" buildStatusProperty="step1BuildStatus" verbose="true" interval="1" timeout="600" />

  <fail message="Failed due to Enterprise build failed">
    <condition>
      <not>
        <equals arg1="${step1BuildStatus}" arg2="OK" /> 
      </not>
    </condition>
  </fail>
</target>
```

You can use the same method to request package, deploy, and other build activities in sequence. This works perfectly as long as each step of the CI build runs independently without needing to know the output of a previous step. However, this isn’t true in the real world. Even in the simple build, package, and deploy process that we described, we see some dependencies. Deploy needs the output of a previous package as input. When you run Deploy manually from the Eclipse menu, you can choose the package that you want to deploy.

The code sample in Listing 2 shows how to get the information from a build result and pass it to another build by using an Ant script. A properties file is created, and the information from the package result is appended to the properties file. This properties file is then used as input to request a deployment.

Listing 2. Get information from a build result and pass it to another build

```xml
<target description="step2" name="step2">
  
  <startBuildActivity ...
```
To find out what properties are needed as input to a Rational Team Concert build, check the **Properties** tab of an existing build result.

**Figure 10. Contents of the Properties tab of a build result**

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>team.deploy.common.packageLabel</td>
<td>team.deploy.common.packageLabel</td>
</tr>
<tr>
<td>team.deploy.common.packageResultUUID</td>
<td>team.deploy.common.packageResultUUID</td>
</tr>
<tr>
<td>team.deploy.common.remotePackageDir</td>
<td>team.deploy.common.remotePackageRootDir/J99e...</td>
</tr>
</tbody>
</table>

**Summary**

The samples and descriptions in this article help you get started with using Rational Team Concert to automate continuous integration of code for your z/OS applications. The key is not only to automate the build and deployment process but to include automated testing also. The second article in this two-part series describes the automated testing capabilities in more detail.