# Developer Zone

Russell Winder

Version 1.0, 2015-10-01

# Table of Contents

# Developer Zone

## Information for GPars Developers

# Build Info

The continuous integration build can be found under:

*Table 1. Build Servers*

| Build Server | Link | Note |
|---|---|---|
| --- | --- | --- |
| Travis-CI | **GPars** | Travis-CI status master |
| Snap-CI | GPars/branch/master | Snap-CI status master |
| JetBrains TeamCity | jetbrains.com | needs registration |

# Issue Tracker

The JIRA issue tracker: http://jira.codehaus.org/browse/GPARS - no longer works.

# Source Repository

The Git repository held at GitHub is the official mainline:

To work on the codebase please fork the repository on GitHub in the usual GitHub workflow way. Keeping the master branch as a mirror of the mainline, working on a feature branch and then sending in pull requests based on that feature branch seems to be the best way of working. Please refer to the Git and GitHub documentation for any further details on using Git and GitHub.

# Mirror Repository

A repository that is a mirror of the GitHub repository was maintained at Codehaus (before they closed) in order to continue integration with various continuous integration servers (over time more of this will migrate to GitHub). Also this Codehaus project was the location of the issue tracker and the route for artifacts to get into the Maven repository.

You should never need to clone this repository, but for completeness, the command:

git clone

creates a clone of the repository in the subdirectory GPars. The above URL gives read-only access to the repository. Those people with write access to the repository should use the URL:

# Personal Clones

Project committers and contributors typically keep their personal clones of the main repository for feature branches:

- Vaclav Pech
- Russel Winder

# Building The Project

The *gradlew* build script will download and setup gradle for the project and execute the build.

# IDE integration

Create an IDEA or Eclipse project files through *gradlew idea* or *gradlew eclipse* commands and you are ready to go.

## IntelliJ IDEA

Upon start or right before building the project, IDEA will prompt you for the JDK version to use. Once you specify that on the project level, you should be good to go.

## The Default IntelliJ IDEA Project File

**GPars** holds a default IDEA project file in the root of the project and is named *GPars_IDEAX.ipr*. This project serves as a master copy for the generated project files (see above) and also to configure our Continuous Integraion. If you, for some reason, decide to use the default project file, you need to go through a few configuraion steps first.

The first time you open the project, you will be prompted to enter a *PROJECT_JDK_NAME* and a *MAVEN_REPOSITORY* variable. These are IDEA variables and not system variables. These results are stored in your home directory.

Each developer can have a unique value. For example, your *MAVEN_REPOSITORY* may be a path on your disk, and *PROJECT_JDK_NAME* can be any string value, e.g. "1.6". This is the name of the Global JDK defined defined within IDEA.

You can setup a global JDK in IDEA under `File→Project Structure→SDKs`. There is a little text box to fill in where you give the JDK a name. Whatever you typed into this textbox is what needs to be typed into the IDEA Environment Variable screen for PROJECT_JDK_NAME.

The *MAVEN_REPOSITORY* variable should point to your local maven repository, which IDEA will be using to keep downloadable artifacts of third-party libraries that are needed to build and run *GPars*. You need to populate the repo first for IDEA to find the necessary artifacts. The best way to do so is to open the */java-demo/pom.xml* file (provided Maven support is enabled in your IDEA installation) and ask IDEA to *Import changes*.

Alternatively you can use the *Refresh* button in the Maven tool window.

Future IDEA environment variables can be declared within the **.ipr** and **.iml** with the syntax $VAR_NAME$. Anything undefined at project startup will prompt the user for entry.

# Code style

If you plan to contribute code to the project, please check out our brief code style guide to make sure your contribution fits seamlessly with the rest of the code base.

# VCS workflow

1. People clone the main GitHub repository
2. People create feature branches in their personal cloned repository
3. People publish their work to possibly cooperate with others on the feature and when ready for review announce the branch asking for people to review. (*git push [mirrorRepo] myFeature*)
4. People reviewing the feature branch fetch the changesets from the public mirror and review running tests (*[git remote add mirrorRepo mirrorRepoUrl;] git fetch [mirrorRepo] myFeature*)
5. If there are no worries about the proposed changes then people say so, where there are issues start a debate on the email list.
6. When changes have been reviewed and agreed, one of the committing authors agrees to merge the branch into their master and pushes to the GitHub main repository (and their public mirror repository of course) (*git checkout master; git pull; git merge --no-ff myFeature;git push*)

Notice the **--no-ff** flag when merging.

Note that this workflow is applicable to all people whether they are committing authors or not. It's just that non-committing authors have to convince a committing author to do the commit. A consequence is that people should not be advised to submit patches on JIRA issues, but instead to specify where their feature branch is so it can be pulled. Obviously patches work as well but the whole point is for everyone to publish their feature branches so others can review them in a VCS context.

# Simplified workflow

Trivial spelling error fixes, extra tests that don't necessitate a change of code but just extend the test coverage, and very simple (non-controversial) bug fixes (with their tests) are currently exempt from having a review process.

Discretion on the part of committing developers is required here. (*git pull; fix; commit; git push*) or (*git pull; git checkout -b myFix; fix; commit; git checkout master; git pull; git merge --no-ff myFix;git push*)

# Upgrading Gradle

1. Install Gradle from an up-to-date Gradle Trunk.
2. Edit the build.gradle file to change the number of the wrapper to the new one.
3. Run `gradle wrapper`
4. If the wrapper is a snapshot, the edit wrapper/gradle-wrapper.properties to add back in the missing snapshots from the repository URL
5. Check the result with `git diff`
6. Check the results with `gradlew clean test`
7. If on Linux, check that the Bamboo build should work with `env -i ./bambooBuild`
8. If everything is successful commit the result `git commit -m ' . . .' -a`
9. Push to the mainline `git push`
10. Push to the personal mirror `git push --mirror . . . `
11. Wait expectantly to see if Bamboo works or not . . .

# The Release Plan

## 1. Set Version

In *build.gradle* and in *doc.properties* set the version property

Also update the **ReleaseNotes.txt** file.

## 2. Write *What's new*

Update the "What's new" section of the user guide as well as the **ReleaseNotes.txt** file.

## 3. Tag the Sources

After a **proper** release, create a tag in the VCS with sources that were used to make the release. Label the tab using the *release-x.x* pattern.

## 4. Build the Project

Issue a full rebuild either for a snapshot or a **proper** release

Make sure all demo programs work

## 5. Upload the Artifacts

Run the Release build plan on Bamboo, which will make all the artifacts available for download.

## 6. Update the Maven Repository

Make sure your repository credentials are in *$USER_HOME/.gradle/gradle.properties* or specify your credentials directly in the *uploadArchives* task in *build.gradle* and add *uploadArchive* task to the desired build task:

Confirm the artifacts have been successfully uploaded for **proper** releases. Within a couple of hours the new **proper** release should be propagated into the maven central repository at http://repo1.maven.org/maven2/org/codehaus/gpars/gpars/.

## 7. Cleanup The Snapshot Repository

After a **proper** release, the older snapshot artifacts should be removed manually from the snapshot repository. Any webdav client, like e.g. AnyClient `http://www.anyclient.com/download.html` should be capable.

# 8. Upload The User Guide and Docs

The generated `User Guide` at */build/docs/manual* should be uploaded to **GPars** .

The javadoc and groovydoc folders should be copied to **GPars** and http://gpars.org/1.2.1/groovydoc/.

# 9. Update The Version

After a **proper** release the version in the build file has to be changed to the next version.

# 10. Update JIRA

**Proper** releases should be also closed in JIRA.

# 11. Tell The World

People are impatiently waiting for the new **GPars** features so now is the time to tell them. New **proper** releases should be announced in the following mailing lists and sites:

- https://groups.google.com/forum/#!forum/gpars-users
- https://groups.google.com/forum/#!forum/gpars-developers
- Any other relevant channel