Development Support: Project Guidelines

From pitawiki

At the end of a project set-up, every project should have...

- URL in SVN
- Jira project
- Jenkins job(s)

Contents

Guidelines for Setting Up a New Maven Project

The Basics

1. Pick a project name.
2. Figure out how your project will break down into modules. Will you have a client? Server? Utility? Each module will have to correspond to a single artifact, for instance a WAR, JAR, POM.
3. Figure out where your project belongs in SVN: shared, programs. Should it be a singular project or is it part of a larger program?
4. Add your project to SVN...
   1. Determine whether to add the project under 'groups' or 'projects'
5. Create a Jenkins job
   1. clean deploy site dependency:analyze
   2. Snapshots trigger
   3. Incremental build
6. Add a trigger to SVN
7. Create Jira project

Guidelines for Setting up the Maven Part

- GroupIds

    - All artifacts within a single project share a groupId
    - The groupId starts with com.tapestrysolutions and include the project name.
    - Programs, which include multiple projects, should include the program and the project name.
        - For example, the Bratwurst program contains 3 sub-projects: Brotchen, Sauerkraut, and Mustard. The respective groupId’s would be:
            - Brotchen: com.tapestrysolutions.bratwurst.brotchen
            - Sauerkraut: com.tapestrysolutions. bratwurst.sauerkraut
            - Mustard:: com.tapestrysolutions. bratwurst.mustard
        - The Snarfblat program only contains the Snarfblat project, so the groupId used would be com.tapestrysolutions.snarfblat.
    - Typically, all modules which share a groupId will be released together. In other words, their versions will match up, they will have only one Jenkins job, and the will all be co-located in SVN. The exception to this is for projects which makes sense to loosely group together. These are usually groupings of utility type projects in which each ‘project’ needs to be released separately, but share common features or purpose.
        - A note on releases: Some people have difficulty with the idea of releasing modules which may not have changed. For instance, project Mustard may contain 3 modules: mustard-yellow,mustard-dijon, and mustard-honey. During a sprint release cycle, only 2 of those modules are worked on (mustard-yellow and mustard-dijon), but during the release all 3 modules are included. So the newest release of the mustard-honey module is actually identical to the previously release version. This approach may seem counter-intuitive to some people, but managing each individual version of each module of each project would be a logistical nightmare. The benefits of grouping modules in projects (and sometimes releasing ‘updated’ versions of modules which have no updates) far outweighs the costs.
    - All groupIds must be unique. Two separate projects should not share a groupId because artifacts are identified by the unique combination of the groupId and artifactId. If projects share groupIds, the are more likely to have artifacts that clobber one another which completely defeats the purpose of Maven’s coordinate system in dependency management.
- ArtifactIds:
    - should repeat the project name, and be followed by a identifier unique to each project
        - For example, the Snarfblat project contains 2 modules: snarfblat-pipe and snarfblat-instrument.
        - In the case of programs with multiple projects, I recommend repeating the program name and the project name along with the unique identifier. For example, bratwurst -mustard-yellow, bratwurst -mustard-dijon, bratwurst -mustard-honey. However, this can make things a bit verbose; especially when following the rule of having your directory match your artifactId exactly. The intent is to make artifacts intuitive and easy to identify. It also can help prevent naming clashes down the road. If two projects contain an artifact simply named ‘client’ you are more likely to run into runtime problems with your classpath than you would if they were named projecta-client and programbeta-projecta-client. This example may seem a little forced, but you will probably encounter this more often than you expect.
    - should use lowercase letters and hyphens only
    - should match exactly the directory they are housed in.
- All projects should inherit an approved parent POM (such as tapestry-parent).
    - This makes it easier to propagate company mandated changes throughout projects such as requirements for licensing headers and standard documentation.
- Intra-project dependencies should use ${project.groupId} and ${project.artifactId}
- Each releasable project, will correspond to a single Jenkins job
    - That Jenkins job will be configured for releases.
- All artifacts deployed to Nexus will originate from Jenkins
    - This insures that no one ever deploys something to Nexus that has not been checked in to SVN.

- Release versions of artifacts will be deployed using the maven-release-plugin and be initiated through Jenkins
  - This insures that the release ‘follows the rules’ of Maven, (such as no SNAPSHOT dependencies) and that all appropriate steps are completed (e.g., SVN tag is created, poms updated appropriately, etc.)
- POMS with type=POM should use ‘parent’ as the unique identifier. For example, a project’ss parent POM would be ‘projectname-parent’, a POM used for logical grouping would be ‘projectname-groupingname-parent’, etc.
- Helpful hint: A good rule of thumb for determining at which level your trunk/branches/tags belong is 1 maven project -> 1 Jira project. If you only have 1 Jira project, you should only have 1 groupId.

Example program/project setups:

Simple project set-up

- projecta
  - trunk
    - projecta-module1
    - projecta-module2
    - projecta-grouping1
      - projecta-grouping1-module1
      - projecta-grouping1-module2
    - projecta-grouping2
      - projecta- grouping2-module1
      - projecta- grouping2-module2
      - pom.xml
    - pom.xml
  - branches
  - tags

Grouping1 is for tidiness and logical organization – no grouping parent pom necessary. Grouping2 is for logical organization and convenience for developers. A pom is defined for the group, and can serve 1 or more functions:

1. Provide a grouping of modules so that developers can build all the grouped modules from a single point. Child modules need not inherit this ‘parent’.
2. Provide an inheritable configuration for a group of modules that share characteristics. For example, all flex components may be grouped under a ‘flex’ parent, which includes configuration for the flexmojos plugin. Note: this setup implies that the child nodes inherit from this parent…however, in most cases the configuration can be kept in the top-most parent and are not needed in a sub-parent.

All modules:

- Inherit from the project parent (or from another POM which inherits from the project’s parent) and share a groupId and version.
- Are released together.

Simple sub-project setup

- projectb
    - trunk
        - projectb-subproject1
            - projectb-subproject1-module1
            - pom.xml
        - projectb-subproject2
            - pom.xml
        - projectb-subproject3
            - projectb-subproject3-module1
            - projectb-subproject3-module2
            - pom.xml
        - projectb -parent
            - pom.xml
    - branches
    - tags

This type of setup is for projects which have a loose coupling and need to be release separately. These are typically utility or framework types of projects in which the projects is really a grouping of similar but somewhat unrelated projects and the modules are typically not used in conjunction with one another. Each sub-project is relatively small (usually consisting of 2 or less modules). These often inherit from a shared parent which is itself managed and released as a sub-project. Each sub-project has its own Jenkins job. Even if the subprojects inherit from a shared parent, they still need to redefine their own SCM info.

Programs

- program1
    - trunk
    - branches
    - tags
- program2
    - program2-projecta
        - trunk
        - branches
        - tags
    - program2-projectb
        - trunk
        - branches
        - tags
    - program1-parent

Programs are high level project groupings. Programs can consist of a single releasable project and be set-up like any of the projects types defined above, or they can contain a collection of projects. A program parent can be defined to add consistency to the configuration of the projects, but is not required.

Guidelines for Versioning your Project

Recommended

- Use 1.0.0-SNAPSHOT type pattern.
- Do not pad with zeros (i.e. 1.8.1 is better than 1.08.01)
- Whenever possible inherit the version from the parent (see Project-level vs Component-level)
- Include project reporting such as FindBugs, CheckStyle, PMD, etc.
- Include SCM element
- Include DistributionManagement with site element
- Include issueTracking element

Project-level vs Component-level

Using project-level versioning makes projects easier to track, update, release, etc. It is always recommended that you keep a single project-level version whenever possible. If you feel that you need component-level versioning, consider first whether your project actually needs to be broken down into multiple projects. The only cases when component-level version makes sense is when the project is really a grouping of similar but somewhat unrelated projects and the modules are not typically used in conjunction with one another. For example, the 'contrib' project contains a collection of distinct sub-projects.

Release Schedules

Recommended Schedule

Projects versioning should follow a pattern similar to one of the following:

- 1.0.0-SNAPSHOT
- 1.0.0
- 1.0.1-SNAPSHOT
- 1.0.1
- 1.1.0-SNAPSHOT
- 1.1.0
- 1.2.0-SNAPSHOT
- ...

The version should increment based on the magnitude of the changes between releases. A good rule of thumb...

- MajorVersion.MinorVersion.IncrementalVersion
- Major Version -- increment when major functionality, or API changes occur
- Minor Version -- increment for new functionality and minor API changes
- Incremental -- increment for bugfix and minor functionality or milestone releases between deliverables

Note: You can also use a shortened version, just be sure to be consistent within a project:

- 1.0-SNAPSHOT
- 1.0
- 1.1-SNAPSHOT
- 1.1

- ...

Alternate Schedule

An alternate schedule can come in handy for teams performing intermediate milestone releases. This alternate schedule would reduce the need of developers to update related dependent projects to reflect each new release:

- 1.0.0-SNAPSHOT
- 1.0.0-beta-1
- 1.0.0-SNAPSHOT
- 1.0.0-beta-2
- 1.0.0
- 1.1.0-SNAPSHOT
- 1.1.0-beta-1
- 1.2.0-SNAPSHOT
- ...

How-To Guides

How To Release a Project

Misc

Adding the Jira plugin.

- Add plugin to reporting section of POM
- Add issueManagement info to POM
- Add username and password to hudson config

Retrieved from "https://pitawiki/wiki/index.php/Development_Support:_Project_Guidelines"
Category: Development Support

- This page was last modified on 2 July 2013, at 21:25.