Plugin Development for Pipeline

Jesse Glick (CloudBees)

Jenkins World

A global DevOps event

2017

Plugin Development for Pipeline

Jesse Glick
CloudBees, Inc.
@tyvole
@jglick on GitHub

Why Support Pipeline?

- centerpiece of Jenkins 2.0, needed for ”CI/CD as code”
- plugin integrations can be leveraged for much richer workflow options
- but a conceptual shift, needs a little extra effort

Agenda

- overview of special requirements for Pipeline compatibility in plugins
- how to convert a plugin working in freestyle to work the same in Pipeline
- developing special support for Pipeline
- higher-level Groovy syntax

How Pipeline Differs

Your workflow as code, not UI

- “freestyle” projects are *static*: configured via UI
- Pipeline `Jenkinsfile` is a *dynamic* script (Groovy DSL)
- so plugin features that perform logic are unnecessary
  - send mail only if the build failed → `post {failure {...}}`
  - expand build env vars in URL → `"http://server/${var}/"`
  - retry up to three times → `retry(3) {...}`
  - anything complicated → helper functions, libraries

Free execution order (part I)

- freestyle projects are not so free; roughly:
  - allocate executor & workspace from (1) agent
  - set up build wrappers
  - check out SCM (or not)
  - run build steps, halting on failure
  - run recorders (0-1 each)
  - tear down wrappers
  - run notifiers (0-1 each)

Free execution order (part II)

- Pipeline projects can do any of the above:
  - in any order
  - never, once, or many times
  - in parallel
  - whether or not previous steps failed
- metadata about steps a *job* ran is not reliably available
  - only what steps a *build* did run in the past (incl. actual args)

Durability & asynchronous activity

- freestyle builds tie up on executor and do not survive Jenkins restart
- Pipeline builds
  - use as many executors as there are `node {...}` blocks active
  - can run across restarts
    - any state must be safely serializable to disk
  - might await user input, external events, etc. indefinitely

Minimum Compliance

https://github.com/jenkinsci/pipeline-plugin/blob/master/DEVGUIDE.md

Jenkins core APIs friendly to Pipeline

- `SimpleBuildStep` (builders, publishers)
- `SimpleBuildWrapper` (wrappers)
- some signatures of `SCM`, `Trigger`, `JobProperty`, etc.
- various core “baselines” needed, typically 1.580.x+
  - good time to use the 2.x Maven parent POM

Removing assumptions

- `AbstractProject` → `Job` (& some specialized interfaces)
- `AbstractBuild` → `Run` (ditto)
- could run on multiple nodes in one build
- could be multiple SCMs checked out in one build
- do not know list of build steps ahead of time
- build as a whole might succeed even if this step failed
- different vars might be in scope at different points in build
- `SimpleBuildWrapper`: state must be `Serializable`

DSL binding & **Pipeline Syntax** integration

- add a `@Symbol`
- `@DataBoundConstructor` for mandatory fields
  - `@DataBoundSetter` for defaultable fields
- nested config: `Describable` with own `@Symbol`
- use Credentials API to manage secrets
- otherwise usual Jelly UI, all interoperable with freestyle

Anticipating “CD-as-code”

- treat parameters as constants
  - (scripts can interpolate variables as needed)
- avoid mandatory global configuration
  - each team operates autonomously using just `Jenkinsfile`
  - share config from folder properties, libraries, `readFile`, …

Testing

Interactive tests

- `mvn hpi:run`
- try: copy **Pipeline Syntax**, paste into `Jenkinsfile`, & run

Automated tests

- use `JenkinsRule` to set up temporary environment
- test deps on `workflow-job`, `workflow-cps`, `workflow-basic-steps`, `workflow-durable-task-step`
- create a `WorkflowJob` w/ a `CpsFlowDefinition`, try running builds
- `StepConfigTester` to check basics of databinding
  - `SnippetizerTester` for advanced checks
- `SemaphoreStep` to simulate `input`, Jenkins restarts, concurrency, &c.

Demo: `SimpleBuildStep` conversion

https://github.com/jglick/wfdev/compare/pipeline

Custom Steps

https://github.com/jenkinsci/workflow-step-api-plugin/blob/master/README.md

Why a custom step?

- use Pipeline-specific APIs (e.g., decorate “flow graph”)
- asynchronous (e.g., `input`)
- wrappers running body >1 times (e.g., `retry`)
- limitations in core interfaces (e.g., env var handling)
- freestyle configuration very inappropriate for Pipeline

Pieces you need

- `Step`: the *definition* of what to run
  - mostly interchangeable with one Groovy function call
- `StepDescriptor`: singleton *kind* of step and its metadata
- `StepExecution`: what is happening at runtime
  - `transient`, `onResume`, `serialVersionUID`, `readResolve`
  - convenience forms for “quick” steps
- `config.jelly`, `help-XXX.html`, `doFillXXXItems`, &c.
  - allows **Pipeline Syntax** to offer “live” examples

Dealing with asynchrony

- `start` method happens in “CPS VM” thread
  - *must be quick*: this is coöperative multitasking
- use background threads for anything else
  - notify the engine when step completes or fails
  - engine notifies you when step is interrupted

Fun with block-scoped steps

- run a body {...} 0, 1, or more times
  - asynch notification when body ends, may return same result
- set environment variables for nested steps
- adjust console output
  - though colors or hyperlinks not supported in Blue Ocean
- define alt. workspaces or pass down any other “context”

Demo: block-scoped step

DSLs & Libraries

Defining global variables

- `GlobalVariable` extension point: predefine symbol in every build
- can have methods & JavaBeans-style properties
  - not like steps: no environment, no asynchronous mode
- may be stateful
- currently requires `workflow-cps` dep; use sparingly

DSL extensions

- some `GlobalVariable` load special DSLs written in Groovy
- generally incompatible with Declarative Pipeline
- *avoid*

Pipeline libraries

- no need to write a plugin at all! share on GitHub
- if “trusted”, can access Jenkins internal APIs
  - or `@Grab` Java libraries
- can be opinionated & complement plugin-provided steps

Jenkins World

A global DevOps event

2017