Build Flow Plugin

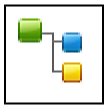
Deprecated: Users should migrate to https://wiki.jenkins-ci.org/display/JENKINS/Pipeline+Plugin

Plugin Information
Distribution of this plugin has been suspended due to unresolved security vulnerabilities, see below.

The current version of this plugin may not be safe to use. Please review the following warnings before use:

- Arbitrary code execution vulnerability

This plugin allows managing Jenkins jobs orchestration using a dedicated DSL, extracting the flow logic from jobs.



Summary

This plugin is designed to handle complex build workflows (aka build pipelines) as a dedicated entity in Jenkins. Without such a plugin, to manage job orchestration the user has to combine parameterized-build, join, downstream-ext and a few more plugins, polluting the job configuration. The build process is then scattered in all those jobs and very complex to maintain. Build Flow enables you to define an upper level Flow item to manage job orchestration and link up rules, using a dedicated DSL. This DSL makes the flow definition very concise and readable.

Configuration

After installing the plugin, you'll get a new Entry in the job creation wizard to create a Flow. Use the DSL editor to define the flow.

Basics

The DSL defines the sequence of jobs to be built :

```
build( "job1" )
build( "job2" )
build( "job3" )
```

You can pass parameters to jobs, and get the resulting AbstractBuild when required :

```
b = build( "job1", param1: "foo", param2: "bar" )
build( "job2", param1: b.build.number )
```

Environment variables from a job can be obtained using the following, which is especially useful for getting things like the checkout revision used by the SCM plugin ('P4_CHANGELIST', 'GIT_REVISION', etc) :

```
def revision = b.environment.get( "GIT_REVISION" )
```

You can also access some pre-defined variables in the DSL :

- "build" the current flow execution
- "out" the flow build console
- "env" the flow environment, as a Map
- "params" triggered parameters
- "upstream" the upstream job, assuming the flow has been triggered as a downstream job for another job.

For example:

```
// output values
out.println 'Triggered Parameters Map:'
out.println params
out.println 'Build Object Properties:'
build.properties.each { out.println "$it.key -> $it.value" }

// use it in the flow
build("job1", parent_param1: params["param1"])
build("job2", parent_workspace:build.workspace)
```

Guard / Rescue

You may need to run a cleanup job after a job (or set of jobs) whenever they succeeded or not. The guard/rescue structure is designed for this use-case. It works mostly like a try+finally block in Java language :

```
guard {
    build( "this_job_may_fail" )
} rescue {
    build( "cleanup" )
}
```

The flow result will then be the worst of the guarded job(s) result and the rescue ones

Ignore

You may also want to just ignore result of some job, that are optional for your build flow. You can use ignore block for this purpose :

```
ignore(FAILURE) {
    build( "send_twitter_notification" )
}
```

The flow will not take care of the triggered build status if it's better than the configured result. This allows you to ignore UNSTABLE < FAILURE < ABORTED

Retry

You can ask the flow to retry a job a few times until success. This is equivalent to the retry-failed-job plugin :

```
retry ( 3 ) {
    build( "this_job_may_fail" )
}
```

Parallel

The flow is strictly sequential, but let you run a set of jobs in parallel and wait for completion. This is equivalent to the join plugin :

```
parallel (
    // job 1, 2 and 3 will be scheduled in parallel.
    { build("job1") },
    { build("job2") },
    { build("job3") }
)
// job4 will be triggered after jobs 1, 2 and 3 complete
build("job4")
```

compared to join plugin, parallel can be used for more complex workflows where the parallel branches can sequentially chain multiple jobs :

```
parallel (
    {
        build("job1A")
        build("job1B")
        build("job1C")
    },
    {
        build("job2A")
        build("job2B")
        build("job2C")
    }
)
```

you also can "name" parallel executions, so you can later use reference to extract parameters / status :

```
join = parallel ([
        first:  { build("job1") },
        second: { build("job2") },
        third:  { build("job3") }
])

// now, use results from parallel execution
build("job4",
       param1: join.first.result.name,
       param2: join.second.lastBuild.parent.name)
```

and this can be combined with other orchestration keywords :

```
parallel (
    {
        guard {
            build("job1A")
        } rescue {
            build("job1B")
        }
    },
    {
        retry 3, {
            build("job2")
        }
    }
)
```

Extension Point

Other plugins that expose themselves to the build flow can be accessed with extension.'plugin-name'

So the plugin foobar might be accessed like:

```
def x = extension.'my-plugin-name'
x.aMethodOnFoobarObject()
```

Plugins implementing extension points

(searching github for "BuildFlowDSLExtension")

- https://github.com/jniesen/build-flow-json-parser-extension-plugin
- https://github.com/dnozay/build-flow-toolbox-plugin
- https://github.com/jenkinsci/external-resource-dispatcher-plugin
- https://github.com/jniesen/build-flow-http-extension-plugin
- https://github.com/jenkinsci/buildflow-extensions-plugin

Implementing Extension

Write the extension in your plugin

```
@Extension(optional = true)
public class MyBuildFlowDslExtension extends BuildFlowDSLExtension {

    /**
     * The extensionName to use for the extension.
     */
    public static final String EXTENSION_NAME = "my-plugin-name";

    @Override
    public Object createExtension(String extensionName, FlowDelegate dsl) {
        if (EXTENSION_NAME.equals(extensionName)) {
            return new MyBuildFlowDsl(dsl);
        }
        return null;
    }
}
```

Write the actual extension

```
public class MyBuildFlowDsl {
    private FlowDelegate dsl;

    /**
     * Standard constructor.
     * @param dsl the delegate.
     */
    public MyBuildFlowDsl(FlowDelegate dsl) {
        this.dsl = dsl;
    }

    /**
     * World.
     */
    public void hello() {
        ((PrintStream)dsl.getOut()).println("Hello World");
    }

}
```

And more ...

future releases may introduce support for some more features and DSL syntax for advanced job orchestration.

Usage

As any Job, the Flow is executed by a trigger, and the Cause is exposed to the flow DSL. If you want to implement a build-pipeline after a commit on your scm, you can configure the flow to be triggered as the first scm-polling job is run, but you can as well use any other trigger (manual trigger, XTrigger plugin, ...) for your flow to integrate in a larger process.

Need help ?

Join jenkins-user mailing list and explain your use-case there

Changelog

Work in Progress

0.20 (release Aug 04, 2016)

- JENKINS-34961 Fix display issues with Jenkins 2.0

0.19 (release May 09, 2016)

- allow a build flow dsl job to be called using **build job:** from a pipeline
- fix dependency with buildgraph-view

0.14 (release Sep. 09, 2014)

- enable test-jar for plugins leveraging the extension point.
- use build.displayName in JobInvocation.toString.

0.13 (release Sep. 09, 2014)

- read DSL from a file
- fix buildgraph when using 2nd level flows.
- swap dependency with buildgraph-view.

0.12 (release May 14, 2014)

- wait for build to be finalized
- fixed-width font in DSL box
- print stack traces when parallel builds fail
- restore ability to **use a workspace**, as a checkbox in flow configuration (useful for SCM using workspace)

0.11.1

- no changes (added the compatibility warning to update center)

0.11 (released Apr. 8, 2014)

- plugin re-licensed to MIT
- **build flow no longer has a workspace**
- Validation of the DSL when configuring the flow
- If a build could not be scheduled show the underlying cause
- extensions can contribute to dsl help
- aborting a flow causes all jobs scheduled and started by the flow to be aborted
- retry is configurable
- misc tweaks to UI and small fixes

0.10

- add support for SimpleParameters (parameter that can be set from a String)
- mechanism to define DSL extensions
- visualization moved to build-graph-view plugin
- minor fixes

0.8 (released Feb. 11, 2013)

- Fix folder support
- Basic flow visualization support (thanks to ~gregory144)
- Alternative map-style way to define parallel executions (thanks to Jeremy Voorhis)

0.7 (released Jan. 11, 2013)

- Add support for ignore(Result)

0.6 (released November 24, 2012)

- Enable "read job" permissions for Anonymous (JENKINS-14027)
- Print errors as .. errors
- Better failed test reporting
- Use transient ref to Job/Build …
- Fix a NullPointer to render FlowCause after jenkins restart
- Use futures for synchronization plus publisher support plus console println cleanup (Pull request #14 from coreyoconnor)
- Call to Parametrized jobs correctly use their default values (Pull request #16 from dbaeli)

0.5 (released September 03, 2012)

- fixed support for publishers (JENKINS-14411)

- improved job configuration UI (dedicated section, help, prepare code mirror integration)
- improved error message

0.4 (released June 28, 2012)

- fixed cast error parsing DSL (Collections$UnmodifiableRandomAccessList' to class 'long') on some version of Jenkins
- add groovy bindings for current build as "build", console output as "out", environment variables Map as "env", and triggered parameters of current build as "params"
- fixed bug when many "Parameters" links were shown for each triggered parameter on build page

0.3 (released April 12, 2012)

- add support for hierarchical project structure (typically : cloudbees folders plugin)

0.2 (released April 9, 2012)

- changed parallel syntax to support nested flows concurrent execution
- fixed serialization issues

0.1 (released April 3, 2012)

- initial release with DSL-based job orchestration