The `build2` Build Bot

Copyright © 2014-2021 the build2 authors (see the AUTHORS file).
Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision `0.14`, October 2021
This revision of the document describes the `build2` build bot `0.14.x` series.

Table of Contents

Preface

This document describes `bbot`, the `build2` build bot. For the build bot command line interface refer to the **`bbot-agent(1)`** and **`bbot-worker(1)`** man pages.

1 Introduction

2 Architecture

The `bbot` architecture includes several layers for security and manageability. At the top we have a `bbot` running in the *controller* mode. The controller monitors various *build sources* for *build tasks*. For example, a controller may poll a `brep` instances for any new packages to built as well as monitor a **`git`** repository for any new commits to test. There can be several layers of controllers with `brep` being just a special kind. A machine running a `bbot` instance in the controller mode is called a *controller host*.

Below the controllers we have a `bbot` running in the *agent* mode normally on Build OS. The agent polls its controllers for *build tasks* to perform. A machine running a `bbot` instance in the agent mode is called a *build host*.

The actual building is performed in the virtual machines and/or containers that are executed on the build host. Inside virtual machines/containers, `bbot` is running in the *worker mode* and receives build tasks from its agent. Virtual machines and containers running a `bbot` instance in the worker mode are collectively called *build machines*.

Let's now examine the workflow in the other direction, that is, from a worker to a controller. Once a build machine is booted (by the agent), the worker inside connects to the TFTP server running on the build host and downloads the *build task manifest*. It then proceeds to perform the build task and uploads the *build result manifest* (which includes build logs) to the TFTP server.

Once an agent receives a build task for a specific build machine, it goes through the following steps. First, it creates a directory on its TFTP server with the *machine name* as its name and places the build task manifest inside. Next, it makes a throw-away snapshot of the build machine and boots it. After booting the build machine, the agent monitors the machine directory on its TFTP server for the build result manifest (uploaded by the worker once the build has completed). Once the result manifest is obtained, the agent shuts down the build machine and discards its snapshot.

To obtains a build task the agent polls via HTTP/HTTPS one or more controllers. Before each poll request the agent enumerates the available build machines and sends this information as part of the request. The controller responds with a build task manifest that identifies a specific build machine to use.

In the task request the agent specifies if only non-interactive, interactive, or both build kinds are supported. If interactive builds are supported, it additionally provides the login information for interactive build sessions. If the controller responds with an interactive build task, then its manifest specifies the breakpoint the worker must stop the task execution at and prompt the user whether to continue or abort the execution. The user can log into the build machine, potentially perform some troubleshooting, and, when done, either answer the prompt or just shutdown the machine.

If the controller has higher-level controllers (for example, `brep`), then it aggregates the available build machines from its agents and polls these controllers (just as an agent would), forwarding build tasks to suitable agents. In this case we say that the *controller act as an agent*. The controller may also be configured to monitor build sources, such as SCM repositories, directly in which case it generates build tasks itself.

In this architecture the build results are propagated up the chain: from a worker, to its agent, to its controller, and so on. A controller that is the final destination of a build result uses email to notify interested parties of the outcome. For example, `brep` would send a notification to the package owner if the build failed. Similarly, a `bbot` controller that monitors a **`git`** repository would send an email to a committer if their commit caused a build failure. The email would include a link (normally HTTP/HTTPS) to the build logs hosted by the controller.

2.1 Configurations

The `bbot` architecture distinguishes between a *machine configuration* and a *build configuration*. The machine configuration captures the operating system, installed compiler toolchain, and so on. The same build machine may be used to "generate" multiple *build configurations*. For example, the same machine can normally be used to produce 32/64-bit and debug/optimized builds.

The machine configuration is *approximately* encoded in its *machine name*. The machine name is a list of components separated with −. Components cannot be empty and must contain only alpha-numeric characters, underscores, dots, and pluses with the whole id being a portably-valid path component.

The encoding is approximate in a sense that it captures only what’s important to distinguish in a particular `bbot` deployment.

The first component normally identifies the operating system and has the following recommended form:

```
[<arch>_][<class>_]<os>[_<version>]
```

For example:

```
windows
windows_10
windows_10.1607
i686_windows_xp
bsd_freebsd_10
linux_centos_6.2
linux_ubuntu_16.04
macos_10.12
```

The second component normally identifies the installed compiler toolchain and has the following recommended form:

```
<id>[<version>][<vendor>][<runtime>]
```

For example:

```
gcc
gcc_6
gcc_6.3
gcc_6.3_mingw_w64
clang_3.9_libc++
clang_3.9_libstdc++
msvc_14
msvc_14u3
icc
```

Some examples of complete machine names:

```
windows_10-msvc_14u3
macos_10.12-clang_10.0
linux_ubuntu_16.04-gcc_6.3
```

Similarly, the build configuration is encoded in a *configuration name* using the same format. As described in Controller Logic, build configurations are generated from machine configurations. As a result, it usually makes sense to have the first component identify the operating systems and the second component – the toolchain with the rest identifying a particular build configuration variant, for example, optimized, sanitized, etc. For example:

```
windows-vc_14-O2
linux-gcc_6-O3_asan
```

2.2 Machine Header Manifest

@@ TODO: need ref to general manifest overview in bpkg, or, better yet, move it to libbutl and ref to that from both places.

The build machine header manifest contains basic information about a build machine on the build host. A list of machine header manifests is sent by `bbot` agents to controllers. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
id: <machine-id>
name: <machine-name>
summary: <string>
```

For example:

```
id: windows_10-msvc_14-1.3
name: windows_10-msvc_14
summary: Windows 10 build 1607 with VC 14 update 3
```

2.2.1 `id`

```
id: <machine-id>
```

The unique machine version/revision/build identifier. For virtual machines this can be the disk image checksum. For a container this can be UUID that is re-generated every time a container filesystem is altered.

Note that we assume that a different machine identifier is assigned on any change that may affect the build result.

2.2.2 `name`

```
name: <machine-name>
```

The machine name.

2.2.3 `summary`

```
summary: <string>
```

The one-line description of the machine.

2.3 Machine Manifest

The build machine manifest contains the complete description of a build machine on the build host (see the Build OS documentation for their origin and location). The machine manifest starts with the machine manifest header with all the header values appearing before any non-header values. The non-header part of manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
type: kvm|nspawn
[mac]: <addr>
[options]: <machine-options>
[changes]: <text>
```

2.3.1 type

```
type: kvm|nspawn
```

The machine type. Valid values are `kvm` (QEMU/KVM virtual machine) and `nspawn` (`systemd-nspawn` container).

2.3.2 mac

```
[mac]: <addr>
```

The fixed MAC address for the machine. Must be in the hexadecimal, comma-separated format. For example:

```
mac: de:ad:be:ef:de:ad
```

If it is not specified, then a random address is generated on the first machine bootstrap which is then reused for each build/re-bootstrap. Note that if you specify a fixed address, then the machine can only be used by a single `bbot` agent.

2.3.3 options

```
[options]: <machine-options>
```

The list of machine options. The exact semantics is machine type-dependent (see below). A single level of quotes (either single or double) is removed in each option before being passed on. Options can be separated with spaces or newlines.

For `kvm` machines, if this value is present, then it replaces the default network and disk configuration when starting the QEMU/KVM hypervisor. The options are pre-processed by replacing the question mark in `ifname=?` and `mac=?` strings with the network interface and MAC address, respectively.

2.3.4 changes

```
[changes]: <text>
```

The description of machine changes in this version.

Multiple `changes` values can be present which are all concatenated in the order specified, that is, the first value is considered to be the most recent. For example:

```
changes: 1.1: initial version
changes: 1.2: increased disk size to 30GB
```

Or:

```
changes:\
1.1
  - initial version

1.2
  - increased disk size to 30GB
  - upgraded bootstrap baseutils
\
```

2.4 Task Manifest

The task manifest describes a build task. It consists of two groups of values. The first group defines the package to build. The second group defines the build configuration to use for building the package. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
name: <package-name>
version: <package-version>
#location: <package-url>
repository-url: <repository-url>
[repository-type]: pkg|git|dir
[trust]: <repository-fp>
[requires]: <package-requirements>
[tests]: <dependency-package>
[examples]: <dependency-package>
[benchmarks]: <dependency-package>
[dependency-checksum]: <checksum>

machine: <machine-name>
target: <target-triplet>
[environment]: <environment-name>
[config]: <config-args>
[host]: true|false
[warning-regex]: <warning-regex>
[interactive]: <breakpoint>
[worker-checksum]: <checksum>
```

2.4.1 `name`

```
name: <package-name>
```

The package name to build.

2.4.2 `version`

```
version: <package-version>
```

The package version to build.

2.4.3 repository-url

```
repository-url: <repository-url>
```

The URL of the repository that contains the package and its dependencies.

2.4.4 repository-type

```
[repository-type]: pkg|git|dir
```

The repository type (see `repository-url` for details). Alternatively, the repository type can be specified as part of the URL scheme. See **bpkg-repository-types(1)** for details.

2.4.5 trust

```
[trust]: <repository-fp>
```

The SHA256 repository certificate fingerprint to trust (see the `bpkg --trust` option for details). This value may be specified multiple times to establish the authenticity of multiple certificates. If the special `yes` value is specified, then all repositories will be trusted without authentication (see the `bpkg --trust-yes` option).

Note that while the controller may return a task with `trust` values, whether they will be used is up to the agent's configuration. For example, some agents may only trust their internally-specified fingerprints to prevent the "man in the middle" attacks.

2.4.6 requires, tests, examples, benchmarks

The primary package manifest values that need to be known by the `bbot` worker before it retrieves the primary package manifest. See Package Manifest for more information on these values.

The controller copies these values from the primary package manifest, except those `tests`, `examples`, and `benchmarks` values which should be excluded from building due to their `builds`, `build-include`, and `build-exclude` manifest values.

2.4.7 dependency-checksum

```
[dependency-checksum]: <checksum>
```

The package dependency checksum received as a part of the previous build task result (see Result Manifest).

2.4.8 `machine`

```
machine: <machine-name>
```

The name of the build machine to use.

2.4.9 `target`

```
target: <target-triplet>
```

The target to build for.

Compared to the autotools terminology, the `machine` value corresponds to `--build` (the machine we are building on) and `target` – to `--host` (the machine we are building for). While we use essentially the same *target triplet* format as autotools for `target`, it is not flexible enough for `machine`.

2.4.10 `environment`

```
[environment]: <environment-name>
```

The name of the build environment to use. See Worker Logic for details.

2.4.11 `config`

```
[config]: <config-args>
```

The additional configuration options and variables. A single level of quotes (either single or double) is removed in each value before being passed to `bpkg`. For example, the following value:

```
config: config.cc.coptions="-O3 -stdlib='libc++'"
```

Will be passed to `bpkg` as the following (single) argument:

```
config.cc.coptions=-O3 -stdlib='libc++'
```

Values can be separated with spaces or newlines. See Controller Logic for details.

2.4.12 `host`

```
[host]: true|false
```

If `true`, then the build configuration is self-hosted. If not specified, `false` is assumed. See Controller Logic for details.

2.4.13 `warning-regex`

```
[warning-regex]: <warning-regex>
```

Additional regular expressions that should be used to detect warnings in the build logs. Note that only the first 512 bytes of each log line is considered.

A single level of quotes (either single or double) is removed in each expression before being used for search. For example, the following value:

```
warning-regex: "warning C4\d{3}: "
```

Will be treated as the following (single) regular expression (with a trailing space):

```
warning C4\d{3}:
```

Expressions can be separated with spaces or newlines. They will be added to the following default list of regular expressions that detect the `build2` toolchain warnings:

```
^warning:
^.+: warning:
```

Note that this built-in list also covers GCC and Clang warnings (for the English locale).

2.4.14 `interactive`

```
[interactive]: <breakpoint>
```

The task execution step to stop at. Can only be present if the agent has specified `interactive-mode` with either the `true` or `both` value in the task request.

The breakpoint can either be a primary step id of the worker script or the special `error` or `warning` value. See Worker Logic for details.

2.4.15 `worker-checksum`

```
[worker-checksum]: <checksum>
```

The worker checksum received as a part of the previous build task result (see Result Manifest).

2.5 Result Manifest

The result manifest describes a build result. The manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
name: <package-name>
version: <package-version>

status:                   <status>
[configure-status]:       <status>
[update-status]:          <status>
[test-status]:            <status>
[install-status]:         <status>
[test-installed-status]: <status>
[uninstall-status]:       <status>

[configure-log]:       <text>
[update-log]:          <text>
[test-log]:            <text>
[install-log]:         <text>
[test-installed-log]: <text>
[uninstall-log]:       <text>

[worker-checksum]:     <checksum>
[dependency-checksum]: <checksum>
```

2.5.1 `name`

```
name: <package-name>
```

The package name from the task manifest.

2.5.2 `version`

```
version: <package-version>
```

The package version from the task manifest.

2.5.3 `status`

```
status: <status>
```

The overall (cumulative) build result status. Valid values are:

```
skip       # Package update and subsequent operations were skipped.
success    # All operations completed successfully.
warning    # One or more operations completed with warnings.
error      # One or more operations completed with errors.
abort      # One or more operations were aborted.
abnormal   # One or more operations terminated abnormally.
```

The `abort` status indicates that the operation has been aborted by `bbot`, for example, because it was consuming too many resources and/or was taking too long. Note that a task can be aborted both by the `bbot` worker as well as the agent. In the later case the whole machine is shut down and no operation-specific status or logs will be included (@@ Maybe we should just include 'log:' with commands that start VM, for completeness?).

The `abnormal` status indicates that the operation has terminated abnormally, for example, due to the package manager or build system crash.

Note that the overall `status` value should appear before any per-operation `*-status` values.

The `skip` status indicates that the received from the controller build task checksums have not changed and the task execution has therefore been skipped under the assumtion that it would have produced the same result. See `agent-checksum`, `worker-checksum`, and `dependency-checksum` for details.

2.5.4 `*-status`

```
[*-status]: <status>
```

The per-operation result status. Note that the `*-status` values should appear in the same order as the corresponding operations were performed and for each `*-status` there should be the corresponding `*-log` value. Currently supported operation names:

```
configure
update
test
install
test-installed
uninstall
```

2.5.5 `*-log`

```
[*-log]: <text>
```

The per-operation result log. Note that the `*-log` values should appear last and in the same order as the corresponding `*-status` values. For the list of supported operation names refer to the `*-status` value description.

2.5.6 `dependency-checksum`

```
[dependency-checksum]: <checksum>
```

The package dependency checksum obtained as a byproduct of the package configuration operation. See **`bpkg-pkg-build(1)`** command's `--rebuild-checksum` option for details.

2.5.7 `worker-checksum`

```
[worker-checksum]: <checksum>
```

The version of the worker logic used to perform the package build task.

2.6 Task Request Manifest

An agent (or controller acting as an agent) sends a task request to its controller via HTTP/HTTPS POST method (@@ URL/API endpoint). The task request starts with the task request manifest followed by a list of machine manifests. The task request manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
agent: <name>
toolchain-name: <name>
toolchain-version: <standard-version>
[interactive-mode]: false|true|both
[interactive-login]: <login>
[fingerprint]: <agent-fingerprint>
```

2.6.1 `agent`

```
agent: <name>
```

The name of the agent host (`hostname`). The name should be unique in a particular `bbot` deployment.

2.6.2 `toolchain-name`

```
toolchain-name: <name>
```

The `build2` toolchain name being used by the agent.

2.6.3 `toolchain-version`

```
toolchain-version: <standard-version>
```

The `build2` toolchain version being used by the agent.

2.6.4 `interactive-mode`

```
[interactive-mode]: false|true|both
```

The agent's capability to perform build tasks only non-interactively (`false`), only interactively (`true`), or both (`both`).

If it is not specified, then the `false` value is assumed.

2.6.5 `interactive-login`

```
[interactive-login]: <login>
```

The login information for the interactive build session. Must be present only if `interactive-mode` is specified with the `true` or `both` value.

2.6.6 `fingerprint`

```
[fingerprint]: <agent-fingerprint>
```

The SHA256 fingerprint of the agent's public key. An agent may be configured not to use the public key-based authentication in which case it does not include this value. However, the controller may be configured to require the authentication in which case it should respond with the 401 (unauthorized) HTTP status code.

2.7 Task Response Manifest

A controller sends the task response manifest in response to the task request initiated by an agent. The response is delivered as a result of the POST method. The task response starts with the task response manifest optionally followed by the task manifest. The task response manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
session: <id>
[challenge]: <text>
[result-url]: <url>
[agent-checksum]: <checksum>
```

2.7.1 `session`

```
session: <id>
```

The identifier assigned to this session by the controller. An empty value indicates that the controller has no tasks at this time in which case all the following values as well as the task manifest are absent.

2.7.2 `challenge`

```
[challenge]: <text>
```

The random, 64 characters long string (nonce) used to challenge the agent's private key. If present, then the agent must sign this string and include the signature in the result request (see below).

The signature should be calculated by encrypting the string with the agent's private key and then `base64`-encoding the result.

2.7.3 `result-url`

```
[result-url]: <url>
```

The URL to POST (upload) the result request to.

2.7.4 `agent-checksum`

```
[agent-checksum]: <checksum>
```

The agent checksum received as a part of the previous build task result request (see Result Request Manifest).

2.8 Result Request Manifest

On completion of a task an agent (or controller acting as an agent) sends the result (upload) request to the controller via the POST method using the URL returned in the task response (see above). The result request starts with the result request manifest followed by the result manifest. Note that there is no result response and only a successful but empty POST result is returned. The result request manifest synopsis is presented next followed by the detailed description of each value in subsequent sections.

```
session: <id>
[challenge]: <text>
[agent-checksum]: <checksum>
```

2.8.1 `session`

```
session: <session-id>
```

The session id as returned by the controller in the task response.

2.8.2 `challenge`

```
[challenge]: <text>
```

The answer to the private key challenge as posed by the controller in the task response. It must be present only if the `challenge` value was present in the task response.

2.8.3 `agent-checksum`

```
[agent-checksum]: <checksum>
```

The version of the agent logic used to perform the package build task.

2.9 Worker Logic

The `bbot` worker builds each package in a *build environment* that is established for a particular build target. The environment has three components: the execution environment (environment variables, etc), build system modules, as well as configuration options and variables.

Setting up of the environment is performed by an executable (script, batch file, etc). Specifically, upon receiving a build task, if it specifies the environment name then the worker looks for the environment setup executable with this name in a specific directory and for the executable called `default` otherwise. Not being able to locate the environment executable is an error.

Once the environment setup executable is determined, the worker re-executes itself as that executable passing to it as command line arguments the target name, the path to the `bbot` worker to be executed once the environment is setup, and any additional options that need to be propagated to the re-executed worker. The environment setup executable is executed in the build directory as its current working directory. The build directory contains the build task `task.manifest` file.

The environment setup executable sets up the necessary execution environment for example by adjusting `PATH` or running a suitable `vcvars` batch file. It then re-executes itself as the `bbot` worker passing to it as command line arguments (in addition to worker options) the list of build system modules (`<env-modules>`) and the list of configuration options and variables (`<env-config-args>`). The environment setup executable must execute the `bbot` worker in the build directory as the current working directory.

The re-executed `bbot` worker then proceeds to test the package from the repository by executing the following commands, collectively called a *worker script*. Each command has a unique *step id* that can be used as a breakpoint as well as a prefix in the `<config-args>`, `<env-config-args>`, and `<env-modules>` values as discussed in Controller Logic. Some step ids have fallback step ids (listed in parenthesis) which are used in the absence of the primary step id values. The <>-values are from the task manifest and the environment.

```
# bpkg.create
#
bpkg -V create <env-modules> <env-config-args> <config-args>

# bpkg.configure.add
#
bpkg -v add <repository-url>
```

```
# bpkg.configure.fetch
#
bpkg -v fetch --trust <repository-fp>

# bpkg.configure.build
#
bpkg -v build --yes --configure-only <package-name>/<package-version>

# bpkg.update
#
bpkg -v update <package-name>

# If the test operation is supported by the package:
#
{
  # bpkg.test
  #
  bpkg -v test <package-name>
}

# For each package referred to by the tests, examples, or benchmarks
# package manifest values not excluded by the bbot controller:
#
{
  # bpkg.test-separate.configure.build (bpkg.configure.build)
  #
  bpkg -v build --yes --configure-only \
       '<package-name> [<version-constraint>]'

  # bpkg.test-separate.update (bpkg.update)
  #
  bpkg -v update <package-name>

  # bpkg.test-separate.test (bpkg.test)
  #
  bpkg -v test <package-name>
}

# If config.install.root is specified:
#
{
  # bpkg.install
  #
  bpkg -v install <package-name>

  # If the package contains subprojects that support the test
  # operation:
  #
  {
    # b.test-installed.create
    #
    b -V create <env-modules> <env-config-args> <config-args>

    # b.test-installed.configure
    #
    b -v configure
```

```
    # b.test-installed.test
    #
    b -v test
  }

  # If any of the tests, examples, or benchmarks package manifest
  # values are not excluded by the bbot controller:
  #
  {
    # bpkg.test-installed.create (bpkg.create)
    #
    bpkg -V create <env-modules> <env-config-args> <config-args>

    # bpkg.test-installed.configure.add (bpkg.configure.add)
    #
    bpkg -v add <repository-url>

    # bpkg.test-installed.configure.fetch (bpkg.configure.fetch)
    #
    bpkg -v fetch --trust <repository-fp>

    # For each package referred to by the tests, examples, or
    # benchmarks package manifest values not excluded by the
    # bbot controller:
    #
    {
      # bpkg.test-separate-installed.configure.build (
      #   bpkg.configure.build)
      #
      bpkg -v build --yes --configure-only \
           '<package-name> [<version-constraint>]'

      # bpkg.test-separate-installed.update (bpkg.update)
      #
      bpkg -v update <package-name>

      # bpkg.test-separate-installed.test (bpkg.test)
      #
      bpkg -v test <package-name>
    }
  }

  # bpkg.uninstall
  #
  bpkg -v uninstall <package-name>
}

# end
#
# This step id can only be used as a breakpoint.
```

For details on configuring and testing installation refer to Controller Logic.

If the package is a build system module, then it is built and tested (using the bundled tests) in a separate configuration that mimics the one used to build `build2` itself. Note that the configuration and environment options and variables are not passed to commands that may affect this configuration. Such commands, therefore, have associated step ids that can only be used as breakpoints (listed in square brackets):

```
# [bpkg.module.create]
#
b -V create config.config.load=~build2
bpkg -v create --existing

# bpkg.module.configure.add (bpkg.configure.add)
#
bpkg -v add <repository-url>

# bpkg.module.configure.fetch (bpkg.configure.fetch)
#
bpkg -v fetch --trust <repository-fp>

# [bpkg.module.configure.build]
#
bpkg -v build --yes --configure-only <package-name>/<package-version>

# [bpkg.module.update]
#
bpkg -v update <package-name>

# If the test operation is supported by the package:
#
{
  # [bpkg.module.test]
  #
  bpkg -v test <package-name>
}
```

If a primary or test package comes from a version control-based repository, then its `dist` meta-operation is also tested as a part of the `bpkg[.*].configure.build` steps by re-distributing the source directory in the load distribution mode after configuration.

If the build is interactive, then the worker pauses its execution at the specified breakpoint and prompts the user whether to continue or abort the execution. If the breakpoint is a step id, then the worker pauses prior to executing every command of the specified step. Otherwise, the breakpoint denotes the result status and the worker pauses if the command results with the specified or more critical status (see Result Manifest).

As an example, the following POSIX shell script can be used to setup the environment for building C and C++ packages with GCC 9 on most Linux distributions.

```
#!/bin/sh

# Environment setup script for C/C++ compilation with GCC 9.
#
# $1  - target
# $2  - bbot executable
# $3+ - bbot options

set -e # Exit on errors.

mode=
case "$1" in
  x86_64-*)
    #mode=-m64
    ;;
  i?86-*)
    mode=-m32
    ;;
  *)
    echo "unknown target: '$1'" 1>&2
    exit 1
    ;;
esac
shift

exec "$@" cc config.c="gcc-9 $mode" config.cxx="g++-9 $mode"
```

2.10 Controller Logic

A `bbot` controller that issues own build tasks maps available build machines (as reported by agents) to *build configurations* according to the `buildtab` configuration file. Blank lines and lines that start with # are ignored. All other lines in this file have the following format:

```
<machine-pattern> <config> <target>[/<environment>] <classes> [<config-arg>]* [<warning-regex>]*

<config-arg> = [<prefix>:](<variable>|<option>)
<prefix> = <tool>[.<phase>][.<operation>[.<command>]]
```

Where `<machine-pattern>` is filesystem wildcard pattern that is matched against available machine names, `<config>` is the configuration name, `<target>` is the build target, optional `<environment>` is the build environment name, `<classes>` is a space-separated list of configuration classes that is matched against the package `builds` values, optional `<config-arg>` list is additional configuration options and variables, and optional `<warning-regex>` list is additional regular expressions that should be used to detect warnings in the logs.

The build configurations can belong to multiple classes with their names reflecting some common configuration aspects, such as the operating system, compiler, build options, etc. Predefined class names are `default`, `all`, `none`, `host`, and `build2`. The default configurations are built by default. A configuration must also belong to the `all` unless it is hidden. A configuration that is self-hosted must also belong to the `host` class and, if it is also self-hosted for build system

modules, to the `build2` class. Valid custom class names must contain only alpha-numeric characters, _, +, -, and `.`, except as the first character for the last three. Class names that start with _ are reserved for the future hidden/special class functionality.

Regular expressions must start with ~, to be distinguished from configuration options and variables. Note that the `<config-arg>` and `<warning-regex>` lists have the same quoting semantics as in the `config` and the `warning-regex` value in the build task manifest. The matched machine name, the target, the environment name, configuration options/variables, and regular expressions are included into the build task manifest.

Values in the `<config-arg>` list can be opionally prefixed with the *step id* or a leading portion thereof to restrict it to a specific step, operation, phase, or tool in the *worker script* (see Worker Logic). Unprefixed values only apply to the `bpkg.create`, `b.test-installed.create`, and `bpkg.test-installed.create` steps. Note that options with values can only be specified using the single argument notation. For example:

```
bpkg:--fetch-timeout=600 bpkg.configure.fetch:--fetch-timeout=60 b:-j1
```

Note that each machine name is matched against every pattern and all the patterns that match produce a configuration. If a machine does not match any pattern, then it is ignored (meaning that this controller is not interested in testing its packages with this machine). If multiple machines match the same pattern, then only a single configuration using any of the machines is produced (meaning that this controller considers these machines equivalent).

As an example, let's say we have a machine named `windows_10-vc_14u3`. If we wanted to test both 32 and 64-bit as well as debug and optimized builds, then we could have generated the following configurations:

```
windows*-msvc_14* windows-msvc_14-32-Z7 i686-microsoft-win32-msvc14.0 "all default msvc i686 debug" config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-32-O2 i686-microsoft-win32-msvc14.0 "all default msvc i686 optimized" config.cc.coptions=/O2 ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-64-Z7 x86_64-microsoft-win32-msvc14.0 "all default msvc x86_64 debug" config.cc.coptions=/Z7 config.cc.loptions=/DEBUG ~"warning C4\d{3}: "
windows*-msvc_14* windows-msvc_14-64-O2 x86_64-microsoft-win32-msvc14.0 "all default msvc x86_64 optimized" config.cc.coptions=/O2 ~"warning C4\d{3}: "
```

As another example, let's say we have `linux_fedora_25-gcc_6` and `linux_ubuntu_16.04-gcc_6`. If all we cared about is testing GCC 6 64-bit builds on Linux, then our configurations could look like this:

```
linux*-gcc_6 linux-gcc_6-g x86_64-linux-gnu "all default gcc debug" config.cc.coptions=-g

linux*-gcc_6 linux-gcc_6-O3 x86_64-linux-gnu "all default gcc optimized" config.cc.coptions=-O3
```

A build configuration class can derive from another class in which case configurations that belong to the derived class are treated as also belonging to the base class (or classes, recursively). The derived and base class names are separated with `:` (no leading or trailing spaces allowed) and the base must be present in the first mentioning of the derived class. For example:

```
linux*-gcc_6 linux-gcc_6-g x86_64-linux-gnu "all gcc-6+ debug" config.cc.coptions=-g

linux*-gcc_6 linux-gcc_6-O3 x86_64-linux-gnu "all gcc-6+ optimized" config.cc.coptions=-O3

linux*-gcc_7 linux-gcc_7-g x86_64-linux-gnu "all gcc-7+:gcc-6+ debug" config.cc.coptions=-g

linux*-gcc_7 linux-gcc_7-O3 x86_64-linux-gnu "all gcc-7+ optimized" config.cc.coptions=-O3
```

A machine pattern consisting of a single `-` is a placeholder entry. Everything about a placeholder is ignored except for the class inheritance information. Note, however, that while all other information is ignored, the configuration name and target must be present but can also be `-`. For example:

```
linux*-gcc_6 linux-gcc_6 x86_64-linux-gnu "all gcc-6+       "
-            -           -                "    gcc-7+:gcc-6+"
linux*-gcc_8 linux-gcc_8 x86_64-linux-gnu "all gcc-8+:gcc-7+"
```

If the `<config-arg>` list contains the `config.install.root` variable that applies to the `bpkg.create` step, then in addition to building and possibly running tests, the `bbot` worker will also test installing and uninstalling each package. Furthermore, if the package contains subprojects that support the test operation and/or refers to other packages via the `tests`, `examples`, or `benchmarks` manifest values which are not excluded by the `bbot` controller, then the worker will additionally build such subprojects/packages against the installation and run their tests (test installed and test separate installed phases).

Two types of installations can be tested: *system* and *private*. A system installation uses a well-known location, such as `/usr` or `/usr/local`, that will be searched by the compiler toolchain by default. A private installation uses a private directory, such as `/opt`, that will have to be explicitly mentioned to the compiler. While the system installation is usually preferable, it may not be always usable because of the potential conflicts with the already installed software, for example, by the system package manager.

As an example, the following two configurations could be used to test system and private installations:

```
linux*-gcc* linux-gcc-sysinstall x86_64-linux-gnu "all default gcc" config.install.root=/usr config.install.sudo=sudo

linux*-gcc* linux-gcc-prvinstall x86_64-linux-gnu "all default gcc" config.install.root=/tmp/install config.cc.poptions=-I/tmp/install/include config.cc.loptions=-L/tmp/install/lib config.bin.rpath=/tmp/install/lib
```

Note also that while building and running tests against the installation the worker makes the `bin` subdirectory of `config.install.root` the first entry in the `PATH` environment variable.