The `build2` Repository Interface

Copyright © 2014-2024 the build2 authors.
Permission is granted to copy, distribute and/or modify this document under the terms of the MIT License.

Revision `0.18`, August 2024
This revision of the document describes the `build2` repository interface `0.18.x` series.

Table of Contents

Preface 1
1 Package Submission 1
1.1 Submission Request Manifest 3
1.2 Submission Result Manifest 3
2 Package CI 4
2.1 CI Request Manifest 6
2.2 CI Overrides Manifest 7
2.3 CI Result Manifest 7
3 Build Artifacts Upload 8
3.1 Upload Request Manifest 10
3.2 Upload Result Manifest 11
4 Package Review Submission 11
4.1 Package Review Manifest 11

Preface

This document describes `brep`, the `build2` package repository web interface. For the command line interface of `brep` utilities refer to the **`brep-load(1)`**, **`brep-clean(1)`**, **`brep-migrate(1)`**, and **`brep-monitor(1)`** man pages.

1 Package Submission

The package submission functionality allows uploading of package archives as well as additional, repository-specific information via the HTTP `POST` method using the `multipart/form-data` content type. The implementation in `brep` only handles uploading as well as basic verification (checksum, duplicates) expecting the rest of the submission and publishing logic to be handled by a separate entity according to the repository policy. Such an entity can be notified by `brep` about a new submission as an invocation of the *handler program* (as part of the HTTP request) and/or via email. It could also be a separate process that monitors the upload data directory.

The submission request without any parameters is treated as the submission form request. If `submit-form` is configured, then such a form is generated and returned. Otherwise, such a request is treated as an invalid submission (missing parameters).

For each submission request `brep` performs the following steps.

1. Verify submission size limit.

   The submission form-data payload size must not exceed `submit-max-size`.

2. Verify the required `archive` and `sha256sum` parameters are present.

   The `archive` parameter must be the package archive upload while `sha256sum` must be its 64 characters SHA256 checksum calculated in the binary mode.

3. Verify other parameters are valid manifest name/value pairs.

   The value can only contain UTF-8 encoded Unicode graphic characters as well as tab (`\t`), carriage return (`\r`), and line feed (`\n`).

4. Check for a duplicate submission.

   Each submission is saved as a subdirectory in the `submit-data` directory with a 12-character abbreviated checksum as its name.

5. Save the package archive into a temporary directory and verify its checksum.

   A temporary subdirectory is created in the `submit-temp` directory, the package archive is saved into it using the submitted name, and its checksum is calculated and compared to the submitted checksum.

6. Save the submission request manifest into the temporary directory.

   The submission request manifest is saved as `request.manifest` into the temporary subdirectory next to the archive.

7. Make the temporary submission directory permanent.

   Move/rename the temporary submission subdirectory to `submit-data` as an atomic operation using the 12-character abbreviated checksum as its new name. If such a directory already exist, then this is a duplicate submission.

8. Invoke the submission handler program.

   If `submit-handler` is configured, invoke the handler program passing to it additional arguments specified with `submit-handler-argument` (if any) followed by the absolute path to the submission directory.

   The handler program is expected to write the submission result manifest to `stdout` and terminate with the zero exit status. A non-zero exit status is treated as an internal error. The handler program's `stderr` is logged.

   Note that the handler program should report temporary server errors (service overload, network connectivity loss, etc.) via the submission result manifest status values in the [500-599] range (HTTP server error) rather than via a non-zero exit status.

   The handler program assumes ownership of the submission directory and can move/remove it. If after the handler program terminates the submission directory still exists, then it is handled by `brep` depending on the handler process exit status and the submission result manifest status value. If the process has terminated abnormally or with a non-zero exit status or the result manifest status is in the [500-599] range (HTTP server error), then the directory is saved for troubleshooting by appending the `.fail` extension followed by a numeric extension to its name (for example, `ff5a1a53d318.fail.1`). Otherwise, if the status is in the [400-499] range (HTTP client error), then the directory is removed. If the directory is left in place by the handler or is saved for troubleshooting, then the submission result manifest is saved as `result.manifest` into this directory, next to the request manifest and archive.

   If `submit-handler-timeout` is configured and the handler program does not exit in the allotted time, then it is killed and its termination is treated as abnormal.

   If the handler program is not specified, then the following submission result manifest is implied:

   ```
   status: 200
   message: package submission is queued
   reference: <abbrev-checksum>
   ```

9. Send the submission email.

   If `submit-email` is configured, send an email to this address containing the submission request manifest and the submission result manifest.

10. Respond to the client.

    Respond to the client with the submission result manifest and its `status` value as the HTTP status code.

Check violations (max size, duplicate submissions, etc) that are explicitly mentioned above are always reported with the submission result manifest. Other errors (for example, internal server errors) might be reported with unformatted text, including HTML.

If the submission request contains the `simulate` parameter, then the submission service simulates the specified outcome of the submission process without actually performing any externally visible actions (e.g., publishing the package, notifying the submitter, etc). Note that the package submission email (`submit-email`) is not sent for simulated submissions.

Pre-defined simulation outcome values are `internal-error-text`, `internal-error-html`, `duplicate-archive`, and `success`. The simulation outcome is included into the submission request manifest and the handler program must at least handle `success` but may recognize additional outcomes.

1.1 Submission Request Manifest

The submission request manifest starts with the below values and in that order optionally followed by additional values in the unspecified order corresponding to the custom request parameters.

```
archive: <name>
sha256sum: <sum>
timestamp: <date-time>
[simulate]: <outcome>
[client-ip]: <string>
[user-agent]: <string>
```

The `timestamp` value is in the ISO-8601 `<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>Z` form (always UTC). Note also that `client-ip` can be IPv4 or IPv6.

1.2 Submission Result Manifest

The submission result manifest starts with the below values and in that order optionally followed by additional values if returned by the handler program. If the submission is successful, then the `reference` value must be present and contain a string that can be used to identify this submission (for example, the abbreviated checksum).

```
status: <http-code>
message: <string>
[reference]: <string>
```

2 Package CI

The CI functionality allows submission of package CI requests as well as additional, repository-specific information via the HTTP `GET` and `POST` methods using the `application/x-www-form-urlencoded` or `multipart/form-data` parameters encoding. The implementation in `brep` only handles reception as well as basic parameter verification expecting the rest of the CI logic to be handled by a separate entity according to the repository policy. Such an entity can be notified by `brep` about a new CI request as an invocation of the *handler program* (as part of the HTTP request) and/or via email. It could also be a separate process that monitors the CI data directory.

The CI request without any parameters is treated as the CI form request. If `ci-form` is configured, then such a form is generated and returned. Otherwise, such a request is treated as an invalid CI request (missing parameters).

For each CI request `brep` performs the following steps.

1. Verify the required `repository` and optional `package` parameters.

   The `repository` parameter is the remote `bpkg` repository location that contains the packages to be tested. If one or more `package` parameters are present, then only the specified packages are tested. If no `package` parameters are specified, then all the packages present in the repository (but excluding complement repositories) are tested.

   Each `package` parameter can specify either just the package name, in which case all the versions of this package present in the repository will be tested, or both the name and version in the `<name>/<version>` form (for example, `libhello/1.2.3`.

2. Verify the optional `overrides` parameter.

   The overrides parameter, if specified, must be the CI overrides manifest upload.

3. Verify other parameters are valid manifest name/value pairs.

   The value can only contain UTF-8 encoded Unicode graphic characters as well as tab (`\t`), carriage return (`\r`), and line feed (`\n`).

4. Generate CI request id and create request directory.

   For each CI request a unique id (UUID) is generated and a request subdirectory is created in the `ci-data` directory with this id as its name.

5. Save the CI request manifest into the request directory.

   The CI request manifest is saved as `request.manifest` into the request subdirectory created on the previous step.

6. Save the CI overrides manifest into the request directory.

   If the CI overrides manifest is uploaded, then it is saved as `overrides.manifest` into the request subdirectory.

7. Invoke the CI handler program.

   If `ci-handler` is configured, invoke the handler program passing to it additional arguments specified with `ci-handler-argument` (if any) followed by the absolute path to the CI request directory.

   The handler program is expected to write the CI result manifest to `stdout` and terminate with the zero exit status. A non-zero exit status is treated as an internal error. The handler program's `stderr` is logged.

   Note that the handler program should report temporary server errors (service overload, network connectivity loss, etc.) via the CI result manifest status values in the [500-599] range (HTTP server error) rather than via a non-zero exit status.

   The handler program assumes ownership of the CI request directory and can move/remove it. If after the handler program terminates the request directory still exists, then it is handled by `brep` depending on the handler process exit status and the CI result manifest status value. If the process has terminated abnormally or with a non-zero exit status or the result manifest status is in the [500-599] range (HTTP server error), then the directory is saved for troubleshooting by appending the `.fail` extension to its name. Otherwise, if the status is in the [400-499] range (HTTP client error), then the directory is removed. If the directory is left in place by the handler or is saved for troubleshooting, then the CI result manifest is saved as `result.manifest` into this directory, next to the request manifest.

   If `ci-handler-timeout` is configured and the handler program does not exit in the allotted time, then it is killed and its termination is treated as abnormal.

   If the handler program is not specified, then the following CI result manifest is implied:

   ```
   status: 200
   message: CI request is queued
   reference: <request-id>
   ```

8. Send the CI request email.

   If `ci-email` is configured, send an email to this address containing the CI request manifest, the potentially empty CI overrides manifest, and the CI result manifest.

9. Respond to the client.

   Respond to the client with the CI result manifest and its `status` value as the HTTP status code.

Check violations that are explicitly mentioned above are always reported with the CI result manifest. Other errors (for example, internal server errors) might be reported with unformatted text, including HTML.

If the CI request contains the `interactive` parameter, then the CI service provides the execution environment login information for each test and stops them at the specified breakpoint.

Pre-defined breakpoint ids are `error` and `warning`. The breakpoint id is included into the CI request manifest and the CI service must at least handle `error` but may recognize additional ids (build phase/command identifiers, etc).

If the CI request contains the `simulate` parameter, then the CI service simulates the specified outcome of the CI process without actually performing any externally visible actions (e.g., testing the package, publishing the result, etc). Note that the CI request email (`ci-email`) is not sent for simulated requests.

Pre-defined simulation outcome values are `internal-error-text`, `internal-error-html`, and `success`. The simulation outcome is included into the CI request manifest and the handler program must at least handle `success` but may recognize additional outcomes.

2.1 CI Request Manifest

The CI request manifest starts with the below values and in that order optionally followed by additional values in the unspecified order corresponding to the custom request parameters.

```
id: <request-id>
repository: <url>
[package]: <name>[/<version>]
[interactive]: <breakpoint>
[simulate]: <outcome>
timestamp: <date-time>
[client-ip]: <string>
[user-agent]: <string>
[service-id]: <string>
[service-type]: <string>
[service-data]: <string>
[service-action]: <action>
```

The `package` value can be repeated multiple times. The `timestamp` value is in the ISO-8601 `<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>Z` form (always UTC). Note also that `client-ip` can be IPv4 or IPv6.

Note that some CI service implementations may serve as backends for third-party services. The latter may initiate CI tasks, providing all the required information via some custom protocol, and expect the CI service to notify it about the progress. In this case the third-party service type as well as optionally the third-party id and custom state data can be communicated to the underlying CI handler program via the respective `service-*` manifest values. Also note that normally a third-party service has all the required information (repository URL,

etc) available at the time of the CI task initiation, in which case the `start` value is specified for the `service-action` manifest value. If that's not the case, the CI task is only created at the time of the initiation without calling the CI handler program. In this case the CI handler is called later, when all the required information is asynchronously gathered by the service. In this case the `load` value is specified for the `service-action` manifest value.

2.2 CI Overrides Manifest

The CI overrides manifest is a package manifest fragment that should be applied to all the packages being tested. The contained values override the whole value groups they belong to, resetting all the group values prior to being applied. Currently, only the following value groups can be overridden:

```
build-email build-{warning,error}-email
builds build-{include,exclude}
*-builds *-build-{include,exclude}
*-build-config
```

For the package configuration-specific build constraint overrides the corresponding configuration must exist in the package manifest. In contrast, the package configuration override (**`*-build-config`**) adds a new configuration if it doesn't exist and updates the arguments of the existing configuration otherwise. In the former case, all the potential build constraint overrides for such a newly added configuration must follow the corresponding **`*-build-config`** override.

Note that the build constraints group values (both common and build package configuration-specific) are overridden hierarchically so that the `[`**`*-`**`]`**`build-`**`{`**`include,exclude`**`}` overrides don't affect the respective `[`**`*-`**`]`**`builds`** values.

Note also that the common and build package configuration-specific build constraints group value overrides are mutually exclusive. If the common build constraints are overridden, then all the configuration-specific constraints are removed. Otherwise, if any configuration-specific constraints are overridden, then for the remaining configurations the build constraints are reset to **`builds: none`**.

See Package Manifest for details on these values.

2.3 CI Result Manifest

The CI result manifest starts with the below values and in that order optionally followed by additional values if returned by the handler program. If the CI request is successful, then the `reference` value must be present and contain a string that can be used to identify this request (for example, the CI request id).

```
status: <http-code>
message: <string>
[reference]: <string>
```

3 Build Artifacts Upload

The build artifacts upload functionality allows uploading archives of files generated as a byproduct of the package builds. Such archives as well as additional, repository-specific information can optionally be uploaded by the automated build bots via the HTTP `POST` method using the `multipart/form-data` content type (see the `bbot` documentation for details). The implementation in `brep` only handles uploading as well as basic actions and verification (build session resolution, agent authentication, checksum verification) expecting the rest of the upload logic to be handled by a separate entity according to the repository policy. Such an entity can be notified by `brep` about a new upload as an invocation of the *handler program* (as part of the HTTP request) and/or via email. It could also be a separate process that monitors the upload data directory.

For each upload request `brep` performs the following steps.

1. Determine upload type.

   The upload type must be passed via the `upload` parameter in the query component of the request URL.

2. Verify upload size limit.

   The upload form-data payload size must not exceed `upload-max-size` specific for this upload type.

3. Verify the required `session`, `instance`, `archive`, and `sha256sum` parameters are present. If `brep` is configured to perform agent authentication, then verify that the `challenge` parameter is also present. See the Result Request Manifest for semantics of the `session` and `challenge` parameters.

   The `archive` parameter must be the build artifacts archive upload while `sha256sum` must be its 64 characters SHA256 checksum calculated in the binary mode.

4. Verify other parameters are valid manifest name/value pairs.

   The value can only contain UTF-8 encoded Unicode graphic characters as well as tab (`\t`), carriage return (`\r`), and line feed (`\n`).

5. Resolve the session.

   Resolve the `session` parameter value to the actual package build information.

6. Authenticate the build bot agent.

Use the `challenge` parameter value and the resolved package build information to authenticate the agent, if configured to do so.

7. Generate upload request id and create request directory.

   For each upload request a unique id (UUID) is generated and a request subdirectory is created in the `upload-data` directory with this id as its name.

8. Save the upload archive into the request directory and verify its checksum.

   The archive is saved using the submitted name, and its checksum is calculated and compared to the submitted checksum.

9. Save the upload request manifest into the request directory.

   The upload request manifest is saved as `request.manifest` into the request subdirectory next to the archive.

10. Invoke the upload handler program.

    If `upload-handler` is configured, invoke the handler program passing to it additional arguments specified with `upload-handler-argument` (if any) followed by the absolute path to the upload request directory.

    The handler program is expected to write the upload result manifest to `stdout` and terminate with the zero exit status. A non-zero exit status is treated as an internal error. The handler program's `stderr` is logged.

    Note that the handler program should report temporary server errors (service overload, network connectivity loss, etc.) via the upload result manifest status values in the [500-599] range (HTTP server error) rather than via a non-zero exit status.

    The handler program assumes ownership of the upload request directory and can move/remove it. If after the handler program terminates the request directory still exists, then it is handled by `brep` depending on the handler process exit status and the upload result manifest status value. If the process has terminated abnormally or with a non-zero exit status or the result manifest status is in the [500-599] range (HTTP server error), then the directory is saved for troubleshooting by appending the `.fail` extension to its name. Otherwise, if the status is in the [400-499] range (HTTP client error), then the directory is removed. If the directory is left in place by the handler or is saved for troubleshooting, then the upload result manifest is saved as `result.manifest` into this directory, next to the request manifest.

    If `upload-handler-timeout` is configured and the handler program does not exit in the allotted time, then it is killed and its termination is treated as abnormal.

If the handler program is not specified, then the following upload result manifest is implied:

```
status: 200
message: <upload-type> upload is queued
reference: <request-id>
```

11. Send the upload email.

    If `upload-email` is configured, send an email to this address containing the upload request manifest and the upload result manifest.

12. Respond to the client.

    Respond to the client with the upload result manifest and its `status` value as the HTTP status code.

Check violations (max size, etc) that are explicitly mentioned above are always reported with the upload result manifest. Other errors (for example, internal server errors) might be reported with unformatted text, including HTML.

3.1 Upload Request Manifest

The upload request manifest starts with the below values and in that order optionally followed by additional values in the unspecified order corresponding to the custom request parameters.

```
id: <request-id>
session: <session-id>
instance: <name>
archive: <name>
sha256sum: <sum>
timestamp: <date-time>

name: <name>
version: <version>
project: <name>
target-config: <name>
package-config: <name>
target: <target-triplet>
[tenant]: <tenant-id>
toolchain-name: <name>
toolchain-version: <standard-version>
repository-name: <canonical-name>
machine-name: <name>
machine-summary: <text>
```

The `timestamp` value is in the ISO-8601 `<YYYY>-<MM>-<DD>T<hh>:<mm>:<ss>Z` form (always UTC).

3.2 Upload Result Manifest

The upload result manifest starts with the below values and in that order optionally followed by additional values if returned by the handler program. If the upload request is successful, then the `reference` value must be present and contain a string that can be used to identify this request (for example, the upload request id).

```
status: <http-code>
message: <string>
[reference]: <string>
```

4 Package Review Submission

4.1 Package Review Manifest

The package review manifest files are per version/revision and are normally stored on the filesystem along with other package metadata (like ownership information). Under the metadata root directory, a review manifest file has the following path:

```
<project>/<package>/<version>/reviews.manifest
```

For example:

```
hello/libhello/1.2.3+2/reviews.manifest
```

Note that review manifests are normally not removed when the corresponding package archive is removed (for example, as a result of a replacement with a revision) because reviews for subsequent versions may refer to review results of previous versions (see below).

The package review file is a manifest list with each manifest containing the below values in an unspecified order:

```
reviewed-by: <string>
result-<name>: pass|fail|unchanged
[base-version]: <version>
[details-url]: <url>
```

For example:

```
reviewed-by: John Doe <john@example.org>
result-build: fail
details-url: https://github.com/build2-packaging/hello/issues/1
```

The `reviewed-by` value identifies the reviewer. For example, a deployment policy may require a real name and email address when submitting a review.

The `result-<name>` values specify the review results for various aspects of the package. At least one result value must be present and duplicates for the same aspect name are not allowed. For example, a deployment may define the following aspect names: `build` (build system), `code` (implementation source code), `test` (tests), `doc` (documentation).

The `result-<name>` value must be one of `pass` (the review passed), `fail` (the review failed), and `unchanged` (the aspect in question hasn't changed compared to the previous version, which is identified with the `base-version` value; see below).

The `base-version` value identifies the previous version on which this review is based. The idea here is that when reviewing a new revision, a patch version, or even a minor version, it is often easier to review the difference between the two versions than to review everything from scratch. In such cases, if some aspects haven't changed since the previous version, then their results can be specified as `unchanged`. The `base-version` value must be present if at least one `result-<name>` value is `unchanged`.

The `details-url` value specifies a URL that contains the details of the review (issues identified, etc). It can only be absent if none of the `result-<name>` values are `fail`.