# Build Server Protocol

## 1. Build Server Protocol

This document is a thought experiment for how a potential Build Server Protocol (BSP) would look like. The document is written by Ólafur Páll Geirsson, Jorge Vicente Cantero with feedback from Guillaume Martres and Eugene Burmako.

The Build Server Protocol is not an official Scala Center project nor is it an approved standard. Everything in this document is subject to change and open for discussions. Including core data structures.

### 1.1. Motivation

The problem this document aims to address is the multiplied effort required to integrate between available Scala language servers and build tools. Currently, every language server must implement custom integrations for the most popular Scala Build tools in order to extract compilation information such as classpaths and source directories. Likewise, new Scala build tools are expected to integrate with all available Scala IDEs. This explosion of integrations is a growing problem due to the recent proliferation of both Scala language servers and build tools supporting Scala.

### 1.2. Background

The Build Server Protocol takes inspiration from the Language Server Protocol (LSP). Unlike in the Language Server Protocol, the language server is referred to as the "client" and a build tool such as sbt/pants/gradle/bazel is referred to as the "server".

The best way to read this document is by considering it as a wishlist from the perspective of an IDE developer. Consider this document as our personal vision for how a bi-directional communication protocol between a build tool and language server could look like.

The code listings in this document are written using Scala syntax. Every data strucuture in this document has a direct translation to JSON and Protobuf. See

Appendix for schema definitions that can be used to automatically generate bindings for different target languages.



## 1.3. Status

A protocol is only worth as much as the quality of the available clients and servers that implement the protocol. A proof-of-concept integration between scalameta/metals and scalacenter/bloop using the Build Server Protocol is in the works. See sbt/sbt#3890 for a discussion on the next steps for adding BSP support in sbt.

The best way to share your thoughts on the Build Server Protocol is to open an issue or pull request to this repository.

## 1.4. Base protocol

The base protocol is identical to the language server base protocol. See https://microsoft.github.io/language-server-protocol/specification for more details.

Like the language server protocol, the build server protocol defines a set of JSON-RPC request, response and notification messages which are exchanged using the base protocol.

## 1.5. Basic Json Structures

In addition to basic data structures in the language server protocol, the build server protocol defines the following additional data structures.

### 1.5.1. Build Target

Build target contains metadata about an artifact (for example library, test, or binary artifact). Using vocabulary of other build tools:

- sbt: a build target is a combined project + config. Example:
  - a regular JVM project with main and test configurations will have 2 build targets, one for main and one for test.
  - a single configuration in a single project that contains both Java and Scala sources maps to one BuildTarget.
  - a project with crossScalaVersions 2.11 and 2.12 containing main and test configuration in each will have 4 build targets.
  - a Scala 2.11 and 2.12 cross-built project for Scala.js and the JVM with main and test configurations will have 8 build targets.
- pants: a pants target corresponds one-to-one with a BuildTarget
- bazel: a bazel target corresponds one-to-one with a BuildTarget

The general idea is that the BuildTarget data structure should contain only information that is is fast or cheap to compute.

```
trait BuildTarget {

  /** The target's unique identifier */
  def id: BuildTargetIdentifier

  /** A human readable name for this target.
    * May be presented in the user interface.
    * Should be unique if possible.
    * The id.uri is used if None. */
  def displayName: Option[String]

  /** The category of this build target. */
```

```
  def kind: List[BuildTargetKind]

  /** The set of languages that this target contains.
    * The ID string for each language is defined in the LSP. */
  def languageIds: List[String]

  /** Language-specific metadata about this target.
    * See ScalaBuildTarget as an example. */
  def data: Option[Json] // Note, matches `any` in the LSP.
}

object BuildTargetKind {
  val Library = 1
  /** This target can be compiled and tested via
    * method buildTarget/test */
  val Test = 2
  /** This target can be tested via method
    * buildTarget/test and may run slower compared to a Test. */
  val IntegrationTest = 3
  /** This target can be run via method buildTarget/run */
  val Main = 4
}
```

#### 1.5.2. Build Target Identifier

A unique identifier for a target.

```
trait BuildTargetIdentifer {
  /** The target's URI */
  def uri: URI
}
```

## 1.6. Actual Protocol

Unlike the language server protocol, the build server protocol does not support dynamic registration of capabilities. The motivation for this change is simplicity. If a motivating example for dynamic registration comes up this decision can be reconsidered. The server and client capabilities must be communicated through the initialize request.

#### 1.6.1. Server Lifetime

Like the language server protocol, the current protocol specification defines that the lifetime of a build server is managed by the client (e.g. a language server

like Dotty IDE). It is up to the client to decide when to start (process-wise) and when to shutdown a server.

#### 1.6.1.1. Initialize Build Request

Like the language server protocol, the initialize request is sent as the first request from the client to the server. If the server receives a request or notification before the initialize request it should act as follows:

- For a request the response should be an error with code: -32002. The message can be picked by the server.
- Notifications should be dropped, except for the exit notification. This will allow the exit of a server without an initialize request.

Until the server has responded to the initialize request with an InitializeBuildResult, the client must not send any additional requests or notifications to the server.

Request:

- method: 'build/initialize'
- params: InitializeBuildParams defined as follows

```
trait InitializeBuildParams {

  /** The rootUri of the workspace */
  def rootUri: DocumentUri

  /** The capabilities of the client */
  def capabilities: BuildClientCapabilities

}

trait BuildClientCapabilities {
    /** The languages that this client supports.
      * The ID strings for each language is defined in the LSP.
      * The server must never respond with build targets for other
      * languages than those that appear in this list. */
    def languageIds: List[String]
}
```

Response:

- result: InitializeBuildResult defined as follows

```
trait InitializeBuildResult {
  /** The capabilities of the build server */
  capabilities: BuildServerCapabilities
```

```
}

trait BuildServerCapabilities {
  /** The server can compile targets via method buildTarget/compile */
  compileProvider: Boolean
  /** The server can test targets via method buildTarget/test */
  testProvider: Boolean
  /** The server can provide a list of targets that contain a
    * single text document via the method textDocument/buildTargets */
  textDocumentBuildTargetsProvider: Boolean
  /** The server provides sources for library dependencies
    * via method buildTarget/dependencySources */
  dependencySourcesProvider: Boolean
  /** The server provides all the resource dependencies
    * via method buildTarget/resources */
  resourcesProvider: Boolean
  /** The server sends notifications to the client on build
    * target change events via buildTarget/didChange */
  buildTargetChangedProvider: Boolean
}
```

#### 1.6.1.2. Initialized Build Notification

Like the language server protocol, the initialized notification is sent from the client to the server after the client received the result of the initialize request but before the client is sending any other request or notification to the server. The server can use the initialized notification for example to initialize intensive computation such as dependency resolution or compilation. The initialized notification may only be sent once.

Notification:

- method: 'build/initialized'
- params: InitializedBuildParams defined as follows

```
trait InitializedBuildParams {

}
```

#### 1.6.1.3. Shutdown Build Request

Like the language server protocol, the shutdown build request is sent from the client to the server. It asks the server to shut down, but to not exit (otherwise the response might not be delivered correctly to the client). There is a separate exit notification that asks the server to exit.

Request:

- method: 'shutdown'
- params: `null`

Response:

- result: `null`
- error: code and message set in case an exception happens during shutdown request.

#### 1.6.1.4. Exit Build Notification

Like the language server protocol, a notification to ask the server to exit its process. The server should exit with success code 0 if the shutdown request has been received before; otherwise with error code 1.

Notification:

- method: 'exit'
- params: `null`

### 1.6.2. Workspace Build Targets Request

The workspace build targets request is sent from the client to the server to ask for the list of all available build targets in the workspace.

Request:

- method: 'workspace/buildTargets'
- params: `WorkspaceBuildTargetsParams`, defined as follows

```
trait WorkspaceBuildTargetsParams {
}
```

Response:

- result: `WorkspaceBuildTargetsResult`, defined as follows

```
trait WorkspaceBuildTargetsResult {
  /** The build targets in this workspace that
    * contain sources with the given language ids. */
  def targets: List[BuildTarget]
}
```

### 1.6.3. Build Target Changed Notification

The build target changed notification is sent from the server to the client to signal a change in a build target. The server communicates during the initialize handshake whether this method is supported or not.

Notification:

- method: 'buildTarget/didChange'
- params: `DidChangeBuildTarget` defined as follows:

```
trait DidChangeBuildTarget {
  def changes: List[BuildTargetEvent]
}
trait BuildTargetEvent {
  /** The identifier for the changed build target */
  def uri: URI
  /** The kind of change for this build target */
  def kind: Option[BuildTargetEventKind]

  /** Any additional metadata about what information changed. */
  def data: Option[Json]
}
object BuildTargetEventKind {
  val Created = 1
  val Changed = 2
  val Deleted = 3
}
```

#### 1.6.4. Build Target Text Documents Request

The build target text documents request is sent from the client to the server to query for the list of source files that are part of a given list of build targets.

- method: `buildTarget/textDocuments`
- params: `BuildTargetTextDocumentsParams`

```
trait BuildTargetTextDocumentsParams {
  def targets: List[BuildTargetIdentifier]
}
```

Response:

- result: `BuildTargetTextDocumentsResponse`, defined as follows

```
trait BuildTargetTextDocumentsResponse {
  /** The source files used by this target */
  def textDocuments: List[TextDocumentIdentifier]
}
```

#### 1.6.5. Text Document Build Targets Request

The text document build targets request is sent from the client to the server to query for the list of targets containing the given text document. The server

communicates during the initialize handshake whether this method is supported or not.

This request may be considered as the inverse of `buildTarget/textDocuments`. This method can be used by a language server on `textDocument/didOpen` to lookup which compiler instance to use to compile that given text document. In the case there are multiple targets (for example different platforms: JVM/JS, or x86/ARM) containing the same source file, the language server may present in the editor multiple options via `textDocument/codeLens` to configure how to dis-ambiguate.

- method: `textDocument/buildTargets`
- params: `TextDocumentBuildTargetsParams`, defined as follows

```
trait TextDocumentBuildTargetsParams {
  def textDocument: TextDocumentIdentifier
}
```

Response:

- result: `TextDocumentBuildTargetsResult`, defined as follows

```
trait TextDocumentBuildTargetsResult {
  def targets: List[BuildTargetIdentifier]
}
```

#### 1.6.6. Dependency Sources Request

The build target dependency sources request is sent from the client to the server to query for the list of sources for the dependency classpath of a given list of build targets. The server communicates during the initialize handshake whether this method is supported or not. This method can be used by a language server on `textDocument/definition` to "Go to definition" from project sources to dependency sources.

- method: `buildTarget/dependencySources`
- params: `DependencySourcesParams`

```
trait DependencySourcesParams {
  def targets: List[BuildTargetIdentifier]
}
```

Response:

- result: `DependencySourcesResult`, defined as follows

```
trait DependencySourcesResult {
  def items: List[DependencySourcesItem]
}
trait DependencySourcesItem {
```

```
  def target: BuildTargetIdentifier
  /** List of resources containing source files of the
    * target's dependencies.
    * Can be source files, jar files, zip files, or directories. */
  def sources: List[URI]
}
```

#### 1.6.7. Resources Request

The build target resources request is sent from the client to the server to query for the list of resources of a given list of build targets. A resource is a data dependency required to be present in the runtime classpath when a build target is run or executed. The server communicates during the initialize handshake whether this method is supported or not. This method can be used by a client

- method: `buildTarget/resources`
- params: `ResourcesParams`

```
trait ResourcesParams {
  def targets: List[BuildTargetIdentifier]
}
```

Response:

- result: `ResourcesResult`, defined as follows

```
trait ResourcesResult {
  def items: List[ResourceItem]
}
trait ResourceItem {
  def target: BuildTargetIdentifier
  /** List of resource files. */
  def resources: List[URI]
}
```

#### 1.6.8. Compile Request

The compile build target request is sent from the client to the server to compile the given list of build targets. The server communicates during the initialize handshake whether this method is supported or not. This method can for example be used by a language server before `textDocument/rename` to ensure that all workspace sources typecheck correctly and are up-to-date.

- method: `buildTarget/compile`
- params: `CompileParams`

```
trait CompileParams {
  def targets: List[BuildTargetIdentifier]
```

```
  /** Optional arguments to the compilation process. */
  def arguments: List[Json]
}
```

Response:

- result: `CompileReport`, defined as follows

```
trait CompileReport {
  def items: List[CompileReportItem]
}
trait CompileReportItem {

  /** The total number of reported errors compiling this target. */
  def errors: Long

  /** The total number of reported warnings compiling the target. */
  def warnings: Long

  /** The total number of milliseconds it took to compile the target. */
  def time: Option[Long]

  /** The total number of lines of code in the given target. */
  def linesOfCode: Option[Long]

}
```

The server is free to send any number of `textDocument/publishDiagnostics` and `window/logMessage` notifications during compilation before completing the response. The client is free to forward these messages to the LSP editor client.

#### 1.6.9. Test Request

The test build target request is sent from the client to the server to test the given list of build targets. The server communicates during the initialize handshake whether this method is supported or not.

- method: `buildTarget/Test`
- params: `TestParams`

```
trait TestParams {
  def targets: List[BuildTargetIdentifier]
  /** Optional arguments to the test execution. */
  def arguments: List[Json]
}
```

Response:

- result: `TestReport`, defined as follows

```
trait TestReport {
  def items: List[TestReportItem]
}
trait TestReportItem {

  /** The compile times before executing tests if the target.
    * An empty field means the target may have already
    * been compiled beforehand. */
  def compileReport: Option[CompileReportItem]

  /** The total number of successful tests. */
  def passed: Long

  /** The total number of failed tests. */
  def failed: Long

  /** The total number of milliseconds it took to run the tests.
    * Should not include compile times. */
  def time: Option[Long]
}
```

The server is free to send any number of `textDocument/publishDiagnostics` and `window/logMessage` notifications during compilation before completing the response. The client is free to forward these messages to the LSP editor client.

## 1.7. Extensions

The build server protocol is designed to be extended with language specific data structures and methods.

### 1.7.1. Scala

The following section contains Scala-specific extensions to the build server protocol.

#### 1.7.1.1. Scala Build Target

`ScalaBuildTarget` is a basic data structure that contains scala-specific metadata for compiling a target containing Scala sources. This metadata is embedded in the `data: Option[Json]` field of the `BuildTarget` definition.

```
trait ScalaBuildTarget {
```

```
  /** The Scala organization that is used for a target. */
  def scalaOrganization: String

  /** The scala version to compile this target */
  def scalaVersion: String

  /** The binary version of scalaVersion.
    * For example, 2.12 if scalaVersion is 2.12.4. */
  def scalaBinaryVersion: String

  /** The target platform for this target */
  def platform: ScalaPlatform

}

object ScalaPlatform {
  val JVM = 1
  val JS = 2
  val Native = 3
}
```

#### 1.7.1.2. Scalac Options Request

The build target scalac options request is sent from the client to the server to query for the list of compiler options necessary to compile in a given list of targets.

- method: `buildTarget/scalacOptions`
- params: `ScalacOptionsParams`

```
trait ScalacOptionsParams {
  def targets: List[BuildTargetIdentifier]
}
```

Response:

- result: `ScalacOptionsResult`, defined as follows

```
trait ScalacOptionsResult {
  def items: List[ScalacOptionsItem]
}
trait ScalacOptionsItem {
    def target: BuildTargetIdentifier
    /** Additional arguments to the compiler.
      * For example, -deprecation. */
    def options: List[String]

    /** The dependency classpath for this target, must be
```

```
      * identical to what is passed as arguments to
      * the -classpath flag in the command line interface
      * of scalac. */
    def classpath: List[String]

    /** The output directory for classfiles produced by this target */
    def classDirectory: String
}
```

#### 1.7.1.3. Scala Test Classes Request

The build target scalac options request is sent from the client to the server to query for the list of fully qualified names of test clases in a given list of targets. This method can for example be used by a language server to attach a "Run test suite" button above the definition of a test suite via `textDocument/codeLens`. To render the code lens, the language server needs to map the fully qualified names of the test targets to the defining source file via `textDocument/definition`. Then, once users click on the button, the language server can pass the fully qualified name of the test class as an argument to the `buildTarget/test` request.

- method: `buildTarget/scalaTestClasses`
- params: `ScalaTestClassesParams`

```
trait ScalaTestClassesParams {
  def targets: List[BuildTargetIdentifier]
}
```

Response:

- result: `ScalaTestClassesResult`, defined as follows

```
trait ScalaTestClassesResult {
  def items: List[ScalaTestClassesItem]
}
trait ScalaTestClassesItem {
    def target: BuildTargetIdentifier
    /** The fully qualified names of the test classes in this target */
    def classes: List[String]
}
```

## 1.8. Appendix

### 1.8.1. Protobuf schema definitions

The data structures presented in this document are accompanied by protobuf schema definitions. See bsp.proto.

### 1.8.2. Scala Bindings

A Scala library implementation of this communication protocol is available in this repository. The public API of this library currently has three direct Scala dependencies:

- ScalaPB - for generation of Scala sources from protobuf schema
- Monix - for asynchronous programming primitives
- Circe - for JSON serialization and parsing of protocol data structures

If there is demand, it should be possible to refactor out all three dependencies to provide a zero dependency core module.