Build Server
Protocol and new IDEAs
Justin @ebenwert

- Build tools engineer at JetBrains
  - I work on the IntelliJ sbt integration
  - I believe in tools before rules
- Obsession: build tools complaints in Gitter
Jorge @jvican

- Devtools for ~2.5 years at Scala Center
  - I co-maintain Scala's incremental compiler (Zinc)
  - I work on build tools and build servers
  - scalac, compiler plugins and infrastructure
- Obsession: developer productivity
Agenda

1. The BSP IDEA
2. The BSP protocol
3. The BSP integrations
Goal

1. Explain why BSP solves a real problem
2. Share our findings with the audience
How BSP came up

...
Use case (I)

Language servers
Use case (II)

Editors

- Vim
- Sublime Text
- Atom
- Visual Studio Code
Build tools
As the ultimate source of truth
100 combinations!
BSP (Build Server Protocol) is an attempt to formalize the communication between language server/editors and build tools.
«**LSP creates the opportunity to reduce the m-times-n complexity problem of providing a high level of support for *any programming language* in any editor, IDE, or client endpoint to a simpler m-plus-n problem.»

-- https://langserver.org/
“BSP creates the opportunity to reduce the m-times-n complexity problem of providing a high level of support for any build tool in any editor, IDE, or client endpoint to a simpler m-plus-n problem.”

-- Justin and Jorge
«Bejeezus, I just want bloody fast and correct compiles for my team.»

-- Sam Halliday, serious devtools engineer
Developer productivity engineers want solutions that are

1. Extensible
2. Easy to maintain
3. And ideally
   1. Build tool independent
   2. Editor independent
--- a/nothing.properties
+++ b/bsp.properties
- build.tool_specific=true
- one.time.effort=false
- shared.code=false
- robust=false
- easier.to.maintain=false
- easier.to.test=false
+ build.tool_specific=false
+ one.time.effort=true
+ shared.code=true
+ robust=true
+ easier.to.maintain=true
+ easier.to.test=true
BSP Protocol
Fundamentals I

1. JSON-RPC-based protocol
2. It has the notion of
   • Request/Response
   • Bidirectional notifications
Fundamentals II

1. Modelled after LSP
   • Specification follows same format
   • Client-driven design
   • It reuses some LSP methods, e.g.
     ▪ window/logMessage
     ▪ textDocument/publishDiagnostics
     ▪ $/cancelRequest

2. Aims to be implementable alongside LSP
Server lifetime

- Firing up BSP server
  - stdin/stdout
  - TCP/UDP connections.
  - Unix Sockets/Windows pipes
- Initializing BSP connection
  - Similar to TCP 3-way handshake
- Shutting down the BSP server
Server lifetime

Request

trait InitializeBuildParams {
  def rootUri: URI
  def capabilities: BuildClientCapabilities
}
trait BuildClientCapabilities {
  def languageIds: List[String]
}
Server lifetime

Response

```
trait InitializeBuildResult {
    capabilities: BuildServerCapabilities
}

trait BuildServerCapabilities {
    compileProvider: Boolean
    testProvider: Boolean
    textDocumentBuildTargetsProvider: Boolean
    dependencySourcesProvider: Boolean
    buildTargetChangedProvider: Boolean
}
```
Server lifetime

Notification

trait InitializedBuildParams {}
Server lifetime

Client \rightarrow Server

build/shutdown

Client \rightarrow Server

build/exit
Core data structure

A common notion of what a target is across different build tools and language servers

```
trait BuildTarget {
    def id: BuildTargetIdentifier
    def displayName: Option[String]
    def languageIds: List[String]
    def data: Option[Json]
}

trait URI { def uri: String }
trait BuildTargetIdentifier {
    def uri: URI
}
```

Build tool module → Build target → IDE module
workspace/buildTargets

Client => Server

trait WorkspaceBuildTargetsParams {}

Server => Client

trait WorkspaceBuildTargetsResult {
  def targets: List[BuildTarget]
}

```scala
trait WorkspaceBuildTargetsParams {}

trait WorkspaceBuildTargetsResult {
  def targets: List[BuildTarget]
}
```
buildTarget/dependencySources

Client => Server

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

Server => Client

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

trait DependencySourcesItem {
  def target: BuildTargetIdentifier
  def sources: List[URI]
}
```
buildTarget/compile

Client => Server

trait CompileParams {
  def targets: List[BuildTargetIdentifier]
  def arguments: List[Json]
}

Server => Client

trait CompileReport {
  def items: List[CompileReportItem]
}
trait CompileReportItem {
  def target: BuildTargetIdentifier
  def errors: Long
  def warnings: Long
  def time: Option[Long]
  def linesOfCode: Option[Long]
}
buildTarget/test

Client => Server

```scala
trait TestParams {
  def targets: List[BuildTargetIdentifier]
  def arguments: List[Json]
}
```

Server => Client

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

trait TestReportItem {
  def target: BuildTargetIdentifier
  def compileReport: Option[CompileReportItem]
  def passed: Long
  def failed: Long
  def ignored: Long
  def time: Option[Long]
}
```
Other BSP methods

Not covered in this presentation, but present in the spec.

- buildTarget/didChange
- buildTarget/dependencyResources
- buildTarget/textDocuments
- textDocument/buildTargets
So... is BSP language agnostic?

Yes!
Meet language extensions

Extensions formalize language-specific metadata, like:

- Which standard library to use.
- Which platform a language runs on.
- Which compilation flags are enabled.
Scala extension

```scala
trait ScalaBuildTarget {
  def scalaOrganization: String
  def scalaCompiler: String
  def scalaVersion: String
  def scalaBinaryVersion: String
  def platform: ScalaPlatform
}

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

Client => Server

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

Server => Client

```scala
trait ScalacOptionsResult {
  def items: List[ScalcOptionItem]
}

trait ScalacOptionsItem {
  def target: BuildTargetIdentifier
  def options: List[String]
  def classpath: List[String]
  def classDirectory: String
}```
Client => Server

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

Server => Client

```scala
trait ScalaTestClassesResult {
  def items: List[ScalaTestClassesItem]
}

trait ScalaTestClassesItem {
  def target: BuildTargetIdentifier
  def classes: List[String]
}
```
On the roadmap

- Add BSP method for file watching.
- Add compile progress notifications.
- Add BSP `buildTarget/run`.
- Enable remote compilation.
  - How do we handle repository state?
    - Pass in diffs like LSP does.
    - Relay repo synchronization to third-party.
On the roadmap

- On the lookout for feedback
  - scalacenter/bsp
- Formal proposal to STP-WG
- Scala/Scala.js-based client integrations:
  - vim
  - vscode
  - sublime/atom
IntelliJ integration
Thanks.

- Do you want to learn more?
  - Come talk to us!
  - Help improve the spec in `scalacenter/bsp`
- Chat on Bloop's Gitter.
- Chat on intellijs-scala's Gitter.