Google

Bazel syncs

Bazel IntelliJ plugin - Public Tech Talk

2021-11-18

Alice Kober-Sotzek

Context

Purpose

- Have a common ground for discussions about speeding up Bazel syncs

Summary

- Bazel syncs consist of various different steps/computations.
- Each step represents an opportunity for improvement. Some would have larger effects than others.
- The devil is in the details.

Request of community

Please consider and share:

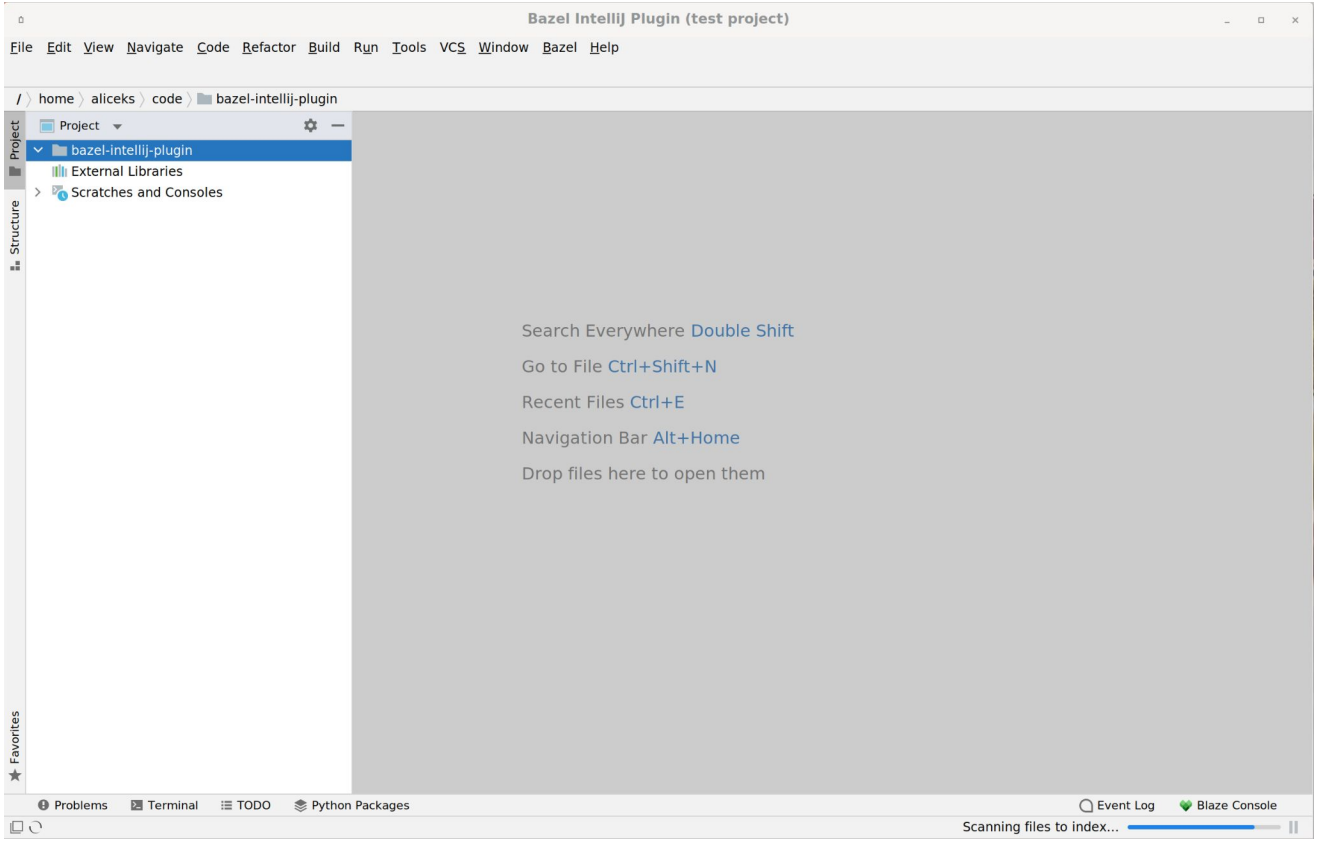
- Which steps do you notice to be especially slow in your setups?
- Which improvements did you try out in your environment? To which effect?

IntelliJ project definition

Project view file: .bazelproject file

```
directories:
 .
 -aswb
 -clwb
 -cpp

targets:
 //ijwb:ijwb_bazel_dev
 //ijwb:ijwb_lib
 //:ijwb_ce_tests
 //:ijwb_ue_tests

workspace_type: intellij_plugin

build_flags:
 --define=ij_product=intellij-latest

test_sources:
 */tests/unittests*
 */tests/integrationtests*
```

Without Bazel sync



Types of Bazel sync

SyncMode

- Special types
  - STARTUP
  - NO_BUILD
- Types involving a build
  - FULL
  - INCREMENTAL
  - PARTIAL

Bazel sync “code pipeline” (preparation, build, post-processing)

- Executed for all Bazel syncs; some parts simply skipped or executed needlessly
- Hooks to provide additional language-specific handling at various stages

STARTUP Bazel sync

- After opening an existing IntelliJ project

- Re-establishes previous in-memory state from disk caches under .cache/JetBrains
  - cache.dat.gz: Outcome of last sync
  - project.view.dat: Read-in project view files

NO_BUILD Bazel sync

Update IntelliJ Project Structure

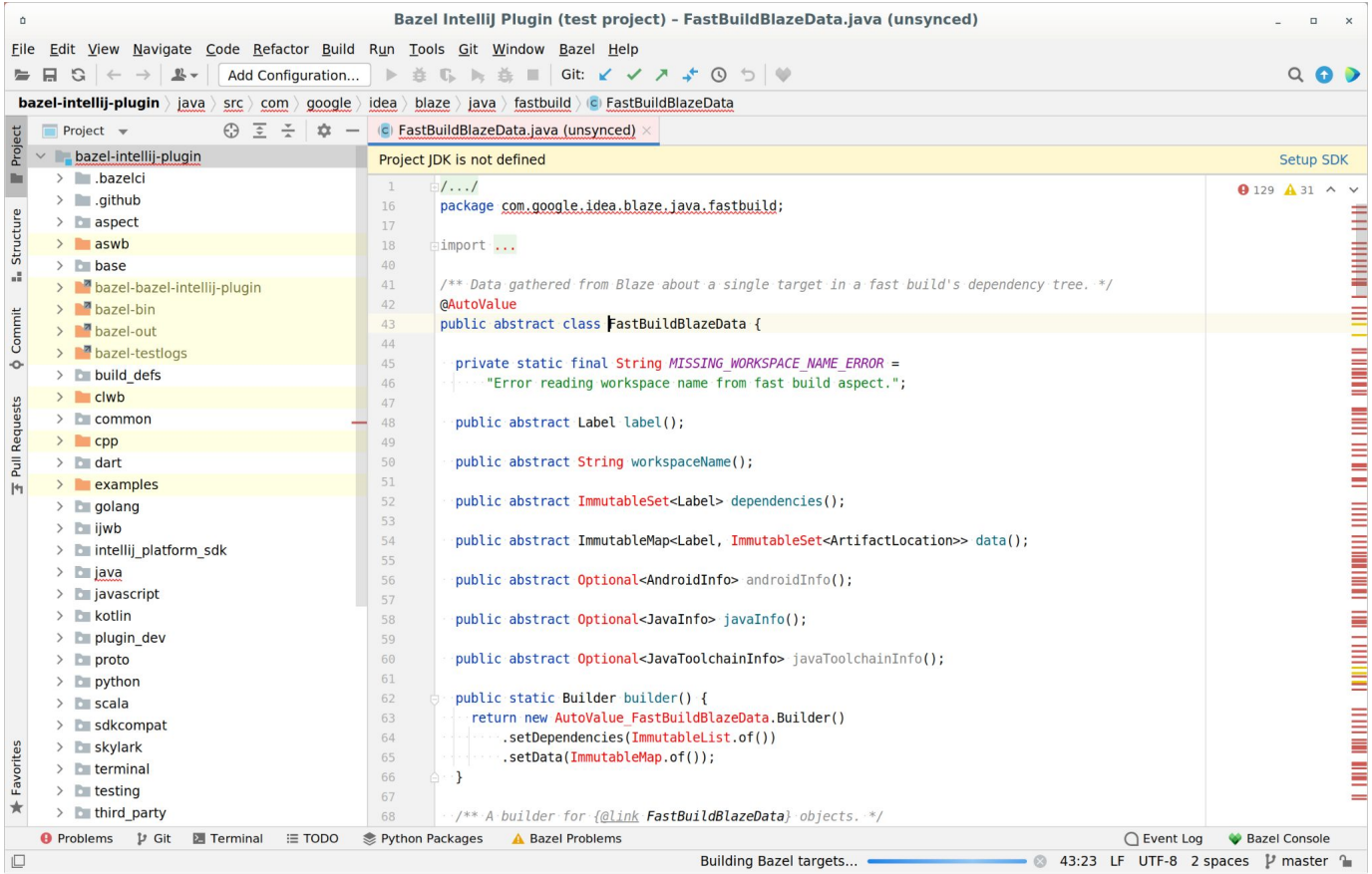
- Project roots (→ directories in .bazelproject).
- One IntelliJ module for all source code.
- Workspace type (java, python, go, …) → IntelliJ module type.
- Add all sub-directories + files below roots.
  - Walk the file system from the roots.
  - Respect excluded directories (prefixed with “-”). → Highlighted with yellow background.
  - Handle and mark test sources separately.

→ Triggers indexing for source code.

NO_BUILD Bazel sync happens before any FULL, INCREMENTAL, PARTIAL Bazel sync.

- Automatically added.
- Ensures correct project structure also on later syncs.

Without Bazel build = After NO_BUILD Bazel sync



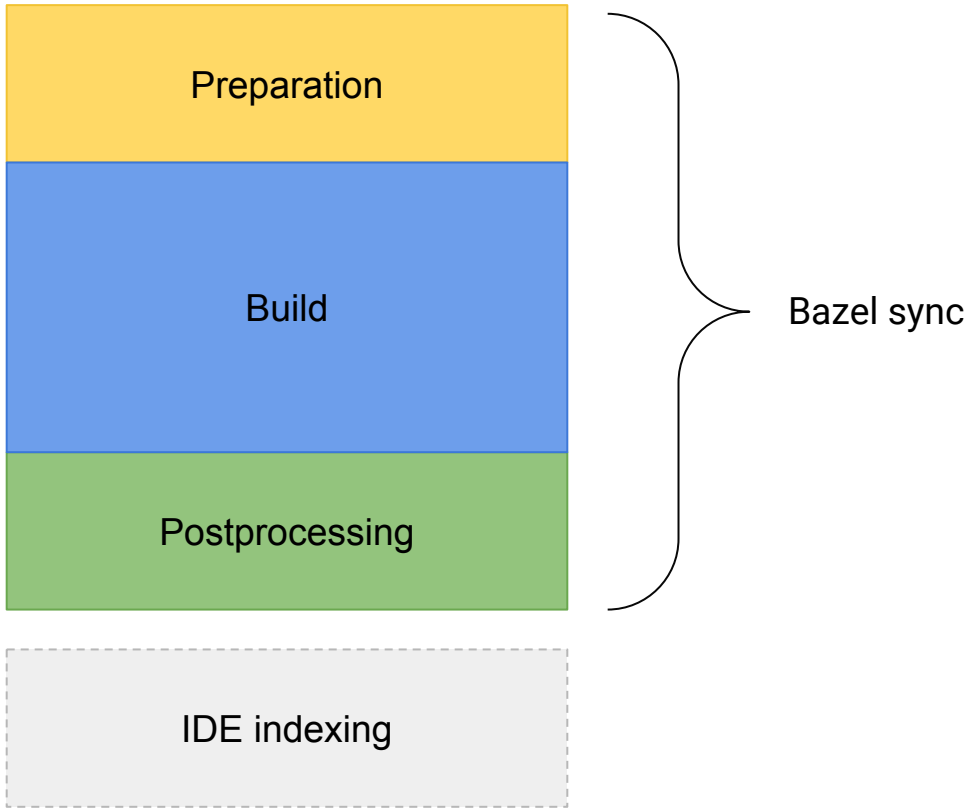
Bazel build - Execution environment

Bazel executions (“query”, “info”, “build”, …) executed on

- Local Bazel server (→ “bazel build”) = *Local build*
  - Advantage: Additional, local caches → Fast incremental builds
  - Much RAM → faster builds; limited RAM → slow builds, potential OOMs
  - Limitation: Just one Bazel execution at one time

- Remote server = *Remote build*
  - Foundations exist in the plugin → Builds can be redirected to a remote machine if desired
  - Advantage: Several builds possible in parallel (self-defined limitation: 10)

Bazel sync - Phases



Preparation - General

- Run Bazel info to learn about directories used by this Bazel server instance
  - Read location of bazel-bin, output_base, bazel-genfiles, …
  - Automatically done for every Bazel sync at the beginning.

- Figure out necessary
  - Aspect (one for all languages)
  - Output groups (different per language)
  - Bazel params (hard-coded e.g. “keep_going” but also from .bazelproject)
  - Targets to build
  - Sharding of targets

Preparation - Determine targets to build

Defined by settings in .bazelproject

- Derived from directories when derive_targets_from_directories is enabled.
- Explicitly defined targets including wildcard patterns (“/...” notation).
- Last step: Remove excluded targets or wildcard patterns (prefixed with “-”).

Derive targets from directories

- via a Bazel query using a heuristic
- filtered to the active languages (→ workspace_type + additional_languages).
- Goal: Build as little as possible compared to manual, wide target definition.

Wildcard expansion of explicitly defined targets

- Only done for user-requested sharding or automatic sharding + remote build
- Transform to non-recursive wildcard targets (prefetch directories + traverse the file system)
- Expand via Bazel query

Preparation - Sharding of targets

Types

- User-requested sharding (shard_sync)
- Automatic sharding (currently always enabled; code for no sharding is still around)

Number of targets per shard

- Default: 1.000 (for user-requested sharding), 10.000 (for automatic sharding)
- Can be user-specified (target_shard_size)
- Limited by system restrictions (e.g. max args on system)

Sharders

- Local build with automatic sharding: Simple partitioning of given target list (retains order)
- Remote build or user-requested sharding: BuildBatchingService on expanded wildcard patterns

Preparation - Sharders = BuildBatchingService

LexicographicTargetSharder

- Simple partitioning on sorted list of targets
- Supports remote + local builds
- For remote builds
  - Only kicks in for larger IntelliJ projects (>= 1000 targets)
  - Spreads workload on all available workers (→ 10 parallel jobs)
  - Given shard size restricted by safe maximum (1000 targets per shard) to avoid OOMs

Custom Sharder

- Simply implement BuildBatchingService and configure this custom sharder
- Could be a remote sharder

Build - Invocation

Inputs

- List of targets
- Our aspect
- Identified output groups (→ influence what is built)
- Bazel params

Failure handling

- keep_going flag → Continue even upon BUILD/compile errors. Different error code for severe Bazel errors (e.g. when OOM).
- Indicate failure to user.
- Surface Bazel warnings/errors in Problems view.
- Use available outputs. → Partially working state for user.

Build - Phases

Loading phase

- Parses, loads, evaluates, and caches BUILD and .bzl files
- Executes macros and builds the target graph

Analysis phase

- Semantic analysis and evaluation of build rules
- Constructs the build dependency graph
- Constructs the action graph → planned schedule of work

Execution phase

- Executes the actions according to the plan
- Re-runs compilation, linting, tools, … as necessary
- Takes the most time of the three phases

Build - Custom aspect

Aspect

- Creates additional actions.
- Walks the build graph, following hardcoded and hand-picked set of language-specific *attributes*.
- Can access internal target info.

For each target on its way, it emits:

- *"Project structure"* file - describes this target (name, type, it's deps & sources, .jars this target contains, how to compile this target, language specifics)
- Collects generated artifacts from this target and all its deps:
  - Java - .jar (hjar or ijar if possible)
  - Others - sources (.h for C++, .go for Golang)

Build - Outputs

- Library jars
  - interface jars (ijar/hjar)
  - Most important: direct dependencies
  - Less priority: source jars
- Jars of generated code
- Language settings (e.g. Java version) and compiler arguments (for other languages like C++)
- Custom artifacts (e.g. aar files for Android; inputs for pre-computed IDE artifacts)

Postprocessing - Retrieval of outputs

- Parse BEP (Build Event Protocol → protobuf messages) output from
  - a file on disk (local build)
  - a special mechanism available for remote builds
- in order to
  - locate build outputs after bazel returns (→ paths to output artifacts)
  - organize artifacts according to targets
  - derive structure of targets. → TargetMap
- Handling of outputs → Post processing
  - Merge with outputs from previous Bazel syncs
  - Prefetch files (e.g. for remote builds)
  - Jar cache enabled: Copy jars to separate area on local disk
  - Remember paths to artifacts
  - “Project structure” artifacts: Cache in own data structures. Includes: TargetMap

Postprocessing - IntelliJ adjustments after Bazel build

Update IntelliJ's project structure

- Update source roots
- Locate and update project SDK (e.g. JDK). Update language level.
- Adjust IntelliJ module
- Configure libraries
  - Attach jars of libraries and generated code
  - Java: jdeps output → Only attach jars of used dependencies of top-level targets
  - Specify as dependencies
- Configure language-specific parts (e.g. IntelliJ facets)

→ Triggers reindexing.

Postprocessing artifact: TargetMap

- Lookup configurations from targets when needed (e.g. locate Go test functions)

- Reverse lookups: SourceToTargetFinder
  - Enables run configurations for tests (source file → test targets)
  - Must be quick
  - Mainly powered by TargetMap

- Beware: SourceToTargetProvider is something else!
  - Purpose: Find targets to add to project for currently uncovered sources.
  - Enabled via: Bazel query

Questions
Comments
Discussion