Building Angular with Bazel

Note: this doc is for developing Angular, it is *not* public documentation for building an Angular application with Bazel.

The Bazel build tool (https://bazel.build) provides fast, reliable incremental builds. We plan to migrate Angular's build scripts to Bazel.

Installation

In order to ensure that everyone builds Angular in a *consistent* way, Bazel will be installed through NPM and therefore it's not necessary to install Bazel manually.

The binaries for Bazel will be provided by the `@bazel/bazelisk` NPM package and its platform-specific dependencies.

You can access Bazel with the `yarn bazel` command

Configuration

The `WORKSPACE` file indicates that our root directory is a Bazel project. It contains the version of the Bazel rules we use to execute build steps, from `npm_bazel_typescript`. The sources on GitHub are published from Google's internal repository (google3).

Bazel accepts a lot of options. We check in some options in the `.bazelrc` file. See the bazelrc doc. For example, if you don't want Bazel to create several symlinks in your project directory (`bazel-*`) you can add the line `build --symlink_prefix=/` to your `.bazelrc` file.

Building Angular

- Build a package: `yarn bazel build packages/core`
- Build all packages: `yarn bazel build packages/...`

You can use ibazel to get a "watch mode" that continuously keeps the outputs up-to-date as you save sources. Note this is new as of May 2017 and not very stable yet.

Testing Angular

- Test package in node: `yarn bazel test packages/core/test:test`
- Test package in karma: `yarn bazel test packages/core/test:test_web`
- Test all packages: `yarn bazel test packages/...`

You can use ibazel to get a "watch mode" that continuously keeps the outputs up-to-date as you save sources.

Various Flags Used For Tests

If you're experiencing problems with seemingly unrelated tests failing, it may be because you're not using the proper flags with your Bazel test runs in Angular.

- `--config=debug` : build and launch in debug mode (see debugging instructions below)
- `--test_arg=--node_options=--inspect=9228` : change the inspector port.
- `--test_tag_filters=<tag>` : filter tests down to tags defined in the `tag` config of your rules in any given `BUILD.bazel`.

Debugging a Node Test

- Open chrome at: chrome://inspect
- Click on `Open dedicated DevTools for Node` to launch a debugger.
- Run test: `yarn bazel test packages/core/test:test --config=debug`

The process should automatically connect to the debugger. For more, see the rules_nodejs Debugging documentation.

For additional info and testing options, see the nodejs_test documentation.

- Click on "Resume script execution" to let the code run until the first `debugger` statement or a previously set breakpoint.
- If you're debugging a test and you want to inspect the generated template instructions, find the template of your component in the call stack and click on `(source mapped from [CompName].js)` at the bottom of the code. You can also disable sourcemaps in the options or go to sources and look into ng:// namespace to see all the generated code.

Debugging a Node Test in VSCode

First time setup:

- Go to Debug > Add configuration (in the menu bar) to open `launch.json`
- Add the following to the `configurations` array:

```
{
  "type": "node",
  "request": "attach",
  "name": "Attach to Remote",
  "port": 9229
}
```

Setting breakpoints directly in your code files may not work in VSCode. This is because the files you're actually debugging are built files that exist in a `./private/...` folder. The easiest way to debug a test for now is to add a `debugger` statement in the code and launch the bazel corresponding test (`yarn bazel test <target> --config=debug`).

Bazel will wait on a connection. Go to the debug view (by clicking on the sidebar or Apple+Shift+D on Mac) and click on the green play icon next to the configuration name (ie `Attach to Remote`).

Debugging a Karma Test

- Run test: `yarn bazel run packages/core/test:test_web_debug` (any `karma_web_test_suite` target has a `_debug` target)
- Open any browser at: http://localhost:9876/debug.html
- Open the browser's DevTools to debug the tests (after, for example, having focused on specific tests via `fit` and/or `fdescribe` or having added `debugger` statements in them)

Debugging Bazel rules

Open `external` directory which contains everything that bazel downloaded while executing the workspace file:

```
open $(yarn -s bazel info output_base)/external
```

See subcommands that bazel executes (helpful for debugging):

```
yarn bazel build //packages/core:package -s
```

To debug nodejs_binary executable paths uncomment `find . -name rollup 1>&2` (~ line 96) in

```
open $(yarn -s bazel info output_base)/external/build_bazel_rules_nodejs/internal/node_launcher.sh
```

Stamping

Bazel supports the ability to include non-hermetic information from the version control system in built artifacts. This is called stamping. You can see an overview at https://www.kchodorow.com/blog/2017/03/27/stamping-your-builds/ In our repo, here is how it's configured:

1. In `tools/bazel_stamp_vars.js` we run the `git` commands to generate our versioning info.
2. In `.bazelrc` we register this script as the value for the `workspace_status_command` flag. Bazel will run the script when it needs to stamp a binary.

Note that Bazel has a `--stamp` argument to `yarn bazel build`, but this has no effect since our stamping takes place in Skylark rules. See https://github.com/bazelbuild/bazel/issues/1054

Remote cache

Bazel supports fetching action results from a cache, allowing a clean build to pick up artifacts from prior builds. This makes builds incremental, even on CI. It works because Bazel assigns a content-based hash to all action inputs, which is used as the cache key for the action outputs. Thanks to the hermeticity property, we can skip executing an action if the inputs hash is already present in the cache.

Of course, non-hermeticity in an action can cause problems. At worst, you can fetch a broken artifact from the cache, making your build non-reproducible. For this reason, we are careful to implement our Bazel rules to depend only on their inputs.

Currently we only use remote caching on CircleCI and we let Angular core developers enable remote caching to speed up their builds.

Remote cache in development

To enable remote caching for your build:

1. Go to the service accounts for the "internal" project
2. Select "Angular local dev", click on "Edit", scroll to the bottom, and click "Create key"
3. When the pop-up shows, select "JSON" for "Key type" and click "Create"
4. Save the key in a secure location
5. Create a file called `.bazelrc.user` in the root directory of the workspace, and add the following content:

```
build --config=angular-team --google_credentials=[ABSOLUTE_PATH_TO_SERVICE_KEY]
```

Remote cache for Circle CI

This feature is experimental, and developed by the CircleCI team with guidance from Angular. Contact Alex Eagle with questions.

How it's configured:

1. In `.circleci/config.yml`, each CircleCI job downloads a proxy binary, which is built from https://github.com/notnoopci/bazel-remote-proxy. The download is done by running `.circleci/setup_cache.sh`. When the feature graduates from experimental, this proxy will be installed by default on every CircleCI worker, and this step will not be needed.
2. Next, each job runs the `setup-bazel-remote-cache` anchor. This starts up the proxy running in the background. In the CircleCI UI, you'll see this step continues running while later steps run, and you can see logging from the proxy process.
3. Bazel must be configured to connect to the proxy on a local port. This configuration lives in `.circleci/bazel.linux.rc` and is enabled because we overwrite the system Bazel settings in /etc/bazel.bazelrc with this file.
4. Each `bazel` command in `.circleci/config.yml` picks up and uses the caching flags.

Diagnosing slow builds

If a build seems slow you can use Bazel to diagnose where time is spent.

The first step is to generate a profile of the build using the `--profile filename_name.profile` flag.

```
yarn bazel build //packages/compiler --profile filename_name.profile
```

This will generate a `filename_name.profile` that you can then analyse using analyze-profile command.

Using the console profile report

You can obtain a simple report directly in the console by running:

```
yarn bazel analyze-profile filename_name.profile
```

This will show the phase summary, individual phase information and critical path.

You can also list all individual tasks and the time they took using `--task_tree`.

```
yarn bazel analyze-profile filename_name.profile --task_tree ".*"
```

To show all tasks that take longer than a certain threshold, use the `--task_tree_threshold` flag. The default behavior is to use a 50ms threshold.

```
yarn bazel analyze-profile filename_name.profile --task_tree ".*" --task_tree_threshold 5000
```

`--task_tree` takes a regexp as argument that filters by the text shown after the time taken.

Compiling TypeScript shows as:

```
70569 ACTION_EXECUTE (10974.826 ms) Compiling TypeScript (devmode) //packages/compiler:compiler []
```

To filter all tasks by TypeScript compilations that took more than 5 seconds, use:

```
yarn bazel analyze-profile filename_name.profile --task_tree "Compiling TypeScript" --task_tree_threshold 5000
```

Using the HTML profile report

A more comprehensive way to visualize the profile information is through the HTML report:

```
yarn bazel analyze-profile filename_name.profile --html --html_details --html_histograms
```

This will generate a `filename_name.profile.html` file that you can open in your browser.

On the upper right corner that is a small table of contents with links to three areas: Tasks, Legend and Statistics.

In the Tasks section you will find a graph of where time is spent. Legend shows what the colors in the Tasks graph mean. Hovering over the background will show what phase that is, while hovering over bars will show more details about that specific action.

The Statistics section shows how long each phase took and how time was spent in that phase. Usually the longest one is the execution phase, which also includes critical path information.

Also in the Statistics section are the Skylark statistic, split in User-Defined and Builtin function execution time. You can click the "self" header twice to order the table by functions where the most time (in ms) is spent.

When diagnosing slow builds you should focus on the top time spenders across all phases and functions. Usually there is a single item (or multiple items of the same kind) where the overwhelming majority of time is spent.

Known issues

Windows

bazel run

If you see the following error:

```
Error: Cannot find module 'C:\users\xxxx\_bazel_xxxx\7lxopdvs\execroot\angular\bazel-out\x64_windows-
fastbuild\bin\packages\core\test\bundling\hello_world\symbol_test.bat.runfiles\angular\c;C:\msys64\users\xxxx\_bazel_xxxx\7lxopdvs\e
out\x64_windows-
fastbuild\bin\packages\core\test\bundling\hello_world\symbol_test.bat.runfiles\angular\packages\core\test\bundling\hello_world\symbo

Require stack:
- internal/preload
    at Function.Module._resolveFilename (internal/modules/cjs/loader.js:793:17)
    at Function.Module._load (internal/modules/cjs/loader.js:686:27)
    at Module.require (internal/modules/cjs/loader.js:848:19)
    at Module._preloadModules (internal/modules/cjs/loader.js:1133:12)
    at loadPreloadModules (internal/bootstrap/pre_execution.js:443:5)
    at prepareMainThreadExecution (internal/bootstrap/pre_execution.js:62:3)
    at internal/main/run_main_module.js:7:1 {
  code: 'MODULE_NOT_FOUND',
  requireStack: [ 'internal/preload' ]
```

`bazel run` only works in Bazel Windows with non-test targets. Ensure that you are using `bazel test` instead.

e.g: `yarn bazel test packages/core/test/bundling/forms:symbol_test`

mkdir missing

If you see the following error::

```
ERROR: An error occurred during the fetch of repository 'npm':
   Traceback (most recent call last):
        File
"C:/users/anusername/_bazel_anusername/idexbm2i/external/build_bazel_rules_nodejs/internal/npm_install/npm_install.bzl", line 618,
column 15, in _yarn_install_impl
                _copy_file(repository_ctx, repository_ctx.attr.package_json)
        File
"C:/users/anusername/_bazel_anusername/idexbm2i/external/build_bazel_rules_nodejs/internal/npm_install/npm_install.bzl", line 345,
column 17, in _copy_file
                fail("mkdir -p %s failed: \nSTDOUT:\n%s\nSTDERR:\n%s" % (dirname, result.stdout, result.stderr))
Error in fail: mkdir -p _ failed:
```

The `msys64` library and associated tools (like `mkdir`) are required to build Angular.

Make sure you have `C:\msys64\usr\bin` in the "system" `PATH` rather than the "user" `PATH`.

After that, a `git clean -xfd`, `yarn`, and `node scripts\build\build-packages-dist.js` should resolve this issue.

Xcode

If you see the following error:

```
$ yarn bazel build packages/...
ERROR: /private/var/tmp/[...]/external/local_config_cc/BUILD:50:5: in apple_cc_toolchain rule @local_config_cc//:cc-compiler-
darwin_x86_64: Xcode version must be specified to use an Apple CROSSTOOL
ERROR: Analysis of target '//packages/core/test/render3:render3' failed; build aborted: Analysis of target '@local_config_cc//:cc-
compiler-darwin_x86_64' failed; build aborted
```

It might be linked to an interaction with VSCode. If closing VSCode fixes the issue, you can add the following line to your VSCode configuration:

```
"files.exclude": {"bazel-*": true}
```

source: https://github.com/bazelbuild/bazel/issues/4603

If VSCode is not the root cause, you might try:

- Quit VSCode (make sure no VSCode is running).

```
bazel clean --expunge
sudo xcode-select -s /Applications/Xcode.app/Contents/Developer
sudo xcodebuild -license
yarn bazel build //packages/core    # Run a build outside VSCode to pre-build the xcode; then safe to run VSCode
```

Source: https://stackoverflow.com/questions/45276830/xcode-version-must-be-specified-to-use-an-apple-crosstool