Styling; A matter of choice

Blaze presents multiple choices for building your site or app's styles. I've taken three of these choices for a deeper look.

The first one, working with Less and Stylus, shows how to work with NPM packages. We could use the same techniques to work with other packages outside the Bazel toolchains, we'll have to test to see if they work.

The other two, PostCSS and SASS show how to work within the Bazel structure using Bazel toolchains and methodoloigy. I haven't decided which one to use for my projects; been thinking about moving away from SASS for a while but I'm not fully convinced that PostCSS is the right solution.

When you're choosing what styling solution to use, think carefully.

Take variables as an example; it's true that both SASS and CSS have variables but they behave differently and each has its advantages and disadvantages.

- SASS variables are static, if you change them you must recompile the stylesheets for the changes to take effect.
- PostCSS uses CSS variables. These variables are live, whenever you change them, all rules using them will be updated and the new result shown to the user.
    - If you define them using the @property syntax, you can define what type of element it is by defining the syntax it uses (either a supported name or the universal syntax definition), an initial value and whether the element inherits down the cascade or not.

Whatever choice you make it will also depend on whether your codebase is already SASS or CSS.

Styling: LESS and Stylus

This buildfile depends on LESS and Stylus to process files into CSS. We first load the Node modules.

```
load("@npm//less:index.bzl", "lessc")
```

```
load("@npm//stylus:index.bzl", "stylus")
```

Gives access to targets defined in the package (but not its subpackages). __pkg__ is a special piece of syntax representing all of the targets in a package. See visibility in the Blaze documentation for more information.

```
package(default_visibility = ["//:__pkg__"])
```

The example uses both LESS and Stylus to generate styles. This uses NPM packages to execute each application. They also use the `args` parameter to handle application specific parameters.

```
lessc(
  name = "base",
  outs = [
    "base.css",
    "base."base.css"  ],
  args = [
    "$(execpath base.less)",
    "$(execpath base.less)"# Output paths must use $(execpath) since Baze
    "--silent",
    "--source-map",
  ],
  data = [
    "base.less",
    "variables.less",
  ],
)

stylus(
  name = "styles",
  outs = [
    "test.css",
    "test.css.map",
  ],
  args = [
```

```
    "$(execpath test.styl)",
    "--out",
    # Output p"--silent"use# Output paths must use $(execpath) since Bazel
    "$(execpath test.css)",
    "--compress",
    "--sourcemap",
  ],
  data = ["test.styl"],
)
```

Styling: PostCSS

Bazel has a ruleset to work with PostCSS, a Javascript toolset to process CSS with Javascript.

Install the `@bazel/postcss` package via NPM.

```
npm i -D @bazel/postcss postcss
```

Add the following block to your `WORKSPACE` to enable PostCSS rules in Bazel:

```
# Loads rules_postcss
http_archive(
  name = "build_bazel_rules_po"build_bazel_rules_postcss"# Make sure to ch
  url = "https://github.com/bazelbuild/rules_postcss/archive/0.5.0.tar.gz"
  strip_prefix = "rules_postcss-0.5.0",
  sha256 = "3f0c754f97e3940ea90f4d6408bfb2aefb3850e7941572b22b1b88579c428f
)
```

`postcss_rules` provide the following rules to use:

- postcss_binary
- postcss_multi_binary
- postcss_plugin
- autoprefixer

For the first example we'll use Autoprefixer on a CSS file that, we assume, was generated earlier on the build process.

As usual, we first load the package and set its visibility. For simplicity's sake I've set it to `tests:__subpackages__` so targets in the test package can see them.

```
load("@build_bazel_rules_postcss//:index.bzl", "autoprefixer")

package(default_visibility = ["//tests:__subpackages__"])
```

Right now, this package has one variable and one target.

The variable (`AUTO_PREFIXER_BROWSERS`) lists the versions of browsers we want to add prefixes for.

The target actually runs Autoprefixer for the browsers specified in the variable.

```
AUTO_PREFIXER_BROWSERS = "ie >= 9, \
edge >= 12, \
firefox >= 42, \
chrome >= 32, \
safari >= 8, \
opera >= 38, \
ios_saf >= 9.2, \
android >= 4.3, \
and_uc >= 9.9"

autoprefixer(
  name = "autoprefixer",
  src = "style.css",
  out = "style_processed.css",
  browsers = AUTO_PREFIXER_BROWSERS,
)
```

The next example will use Autoprefixer as an NPM plugin instead of the Bazel-bound one. Using this technique we can use any of the extensive PostCSS plugin collection in your projects.

The downside is that each plugin must have a corresponding `postcss_plugin` entry in the BUILD file, otherwise Bazel will not be happy and will fail the build.

```
load("//:index.bzl", "postcss_multi_binary", "postcss_plugin")

package(default_visibility = ["//tests:__subpackages__"])

AUTO_PREFIXER_BROWSERS = "ie >= 9, \
edge >= 12, \
firefox >= 42, \
chrome >= 32, \
safari >= 8, \
opera >= 38, \
ios_saf >= 9.2, \
android >= 4.3, \
and_uc >= 9.9"

postcss_plugin(
  name = "autoprefixer",
  node_require = "autoprefixer",
  deps = ["@npm//autoprefixer"],
)

postcss_binary(
  name = "styles",
  src = "styles.css",
  plugins = {
    ":autoprefixer": "[{ browsers: '%s' }]" % AUTO_PREFIXER_BROWSERS,
  },
)
```

Styling: SASS/SCSS

Lately I've been ambivalent about SASS because of their switching the main distribution to Dart and deprecating both Ruby SASS and LibSASS, while LibSASS will receive maintenance releases indefinitely, there will be no new features.

Since the SASS lead developer works at Google, it's unsurprising that there is a Blaze set of Rules for SASS.

To load the rules, add the folowing to your WORKSPACE file.

In addition to loading the rules we also load transitive dependencies for SASS and configure the repositories necessary for the SASS rules.

```
http_archive(
  name = "io_bazel_rules_sass",
  # Make sure to check for the latest version when you install
  url = "https://github.com/bazelbuild/rules_sass/archive/1.26.3.zip",
  sha256 = "9dcfba04e4af896626f4760d866f895ea"https://github.com/bazelbui
)

# Fetch required transitive dependencies.
load("@io_bazel_rules_sass//:package.bzl", "rules_sass_dependencies")
rules_sass_dependencies()

"@io_bazel_rules_sass//:package.bzl"# Setup sass rules repositories
load("@io_bazel_rules_sass//:defs.bzl", "sass_repositories")
sass_repositories()
```

The rest of this section assumes the following directory structure. The WORKSPACE file is included above `my_project` at the root of the repository.

Each directory in this project has its own BUILD file that we'll discuss later in the post.

```
my_project/
  hello_world/
    BUILD
    main.scss
  shared/
    BUILD
    _fonts.scss
    _colors.scss
```

The structure is important. If `hello_world` and `shared` are not in the root of the project, Bazel will not find the BUILD files and exit with an error. I'm researching how to change this so it'll work in subdirectories

`hello_world/main.scss` is simple. It imports the partial files from the shared directory, `fonts` and `colors` and uses variable definitions from those files to construct the styles rules.

```
@import "shared"shared/fonts"@import "shared/colors";

html {
  body {
    font-family: $default-font-stack;
    h1 {
      font-family: $modern-font-stack;
      color: $example-red;
    }
  }
}
```

The BUILD file inside the `hello_world` directory is simple.

It makes the visibility of the BUILD file public and it loads the sass_binary definition.

```
package(default_visibility = ["//visibility:public"])

load("@io_bazel_rules_sass//:defs.bzl", "sass_binary")
```

It then creates a SASS file with both colors and fonts partials as dependencies.

```
sass_binary(
  name = "hello_world",
  src = "main.scss",
  deps = [
    "//shared:colors",
    "//shared:fonts",
```

```
  ],
)
```

The shared directory has files that may be used in multiple locations. For this example the partials (files starting with an underscore _)

There is one for fonts (`_fonts.scss`):

```
$default-font-stack: Cambria, "Hoefler Text", serif;
$modern-font-stack: Constantia, "Lucida Bright", serif;
```

And one for colors (`_colors.scss`):

```
$example-blue: #0000ff;
$example-red: #ff0000;
```

We then use the `sass_library` rule to tell Blaze that these are library files and that are used as reference, without them generating an output file on their own.

As usual we set a default visibility and load the `sass_library` rule

```
package(default_visibility = ["//visibility:public"])

load("@io_bazel_rules_sass//:defs.bzl", "sass_library")
```

Each partial file must be registered with it's own `sass_library` rule, each with its own attributes

```
sass_library(
  name = "colors",
  srcs = ["_colors.scss"],
)

sass_library(
  name = "fonts",
```

```
  srcs = ["_fonts.scss"],
)
```