Angular CLI builders

Contents ›

CLI builders

Builder project structure

Creating a builder

A number of Angular CLI commands run a complex process on your code, such as linting, building, or testing. The commands use an internal tool called Architect to run *CLI builders*, which apply another tool to accomplish the desired task.

With Angular version 8, the CLI Builder API is stable and available to developers who want to customize the Angular CLI by adding or modifying commands. For example, you could supply a builder to perform an entirely new task, or to change which third-party tool is used by an existing command.

This document explains how CLI builders integrate with the workspace configuration file, and shows how you can create your own builder.

> You can find the code from the examples used here in this GitHub repository.

CLI builders

The internal Architect tool delegates work to handler functions called *builders*. A builder handler function receives two arguments; a set of input `options` (a JSON object), and a `context` (a `BuilderContext` object).

The separation of concerns here is the same as with schematics, which are used for other CLI commands that touch your code (such as `ng generate`).

- Options are given by the CLI user, context is provided by and provides access to the CLI Builder API, and the developer provides the behavior.
- The `BuilderContext` object provides access to the scheduling method, `BuilderContext.scheduleTarget()`. The scheduler executes the builder handler function with a given target configuration.

The builder handler function can be synchronous (return a value) or asynchronous (return a Promise), or it can watch and return multiple values (return an Observable). The return value or values must always be of type `BuilderOutput`. This object contains a Boolean `success` field and an optional `error` field that can contain an error message.

Angular provides some builders that are used by the CLI for commands such as `ng build`, `ng test`, and `ng lint`. Default target configurations for these and other built-in CLI builders can be found (and customized) in the "architect" section of the workspace configuration file, `angular.json`. You can also extend and customize Angular by creating your own builders, which you can run using the `ng run` CLI command.

Builder project structure

A builder resides in a "project" folder that is similar in structure to an Angular workspace, with global configuration files at the top level, and more specific configuration in a source folder with the code files that define the behavior. For example, your `myBuilder` folder could contain the following files.

FILES	PURPOSE
`src/my-builder.ts`	Main source file for the builder definition.
`src/my-builder.spec.ts`	Source file for tests.
`src/schema.json`	Definition of builder input options.
`builders.json`	Builders definition.
`package.json`	Dependencies. See https://docs.npmjs.com/files/package.json.
`tsconfig.json`	TypeScript configuration.

You can publish the builder to `npm` (see Publishing your Library). If you publish it as `@example/my-builder`, you can install it using the following command.

```
npm install @example/my-builder
```

Creating a builder

As an example, let's create a builder that executes a shell command. To create a builder, use the `createBuilder()` CLI Builder function, and return a `Promise<BuilderOutput>` object.

src/my-builder.ts (builder skeleton)

```
import { BuilderContext, BuilderOutput, createBuilder } from '@angular-
devkit/architect';
import { JsonObject } from '@angular-devkit/core';

interface Options extends JsonObject {
  command: string;
```

```
  args: string[];
}

export default createBuilder(commandBuilder);

function commandBuilder(
  options: Options,
  context: BuilderContext,
  ): Promise<BuilderOutput> {
}
```

Now let's add some logic to it. The following code retrieves the command and arguments from the user options, spawns the new process, and waits for the process to finish. If the process is successful (returns a code of 0), it resolves the return value.

src/my-builder.ts (builder)

```
import { BuilderContext, BuilderOutput, createBuilder } from '@angular-
devkit/architect';
import { JsonObject } from '@angular-devkit/core';
import * as childProcess from 'child_process';

interface Options extends JsonObject {
  command: string;
  args: string[];
}

export default createBuilder(commandBuilder);

function commandBuilder(
  options: Options,
  context: BuilderContext,
  ): Promise<BuilderOutput> {
    const child = childProcess.spawn(options.command, options.args);
    return new Promise(resolve => {
      child.on('close', code => {
        resolve({ success: code === 0 });
      });
    });
}
```

Handling output

By default, the `spawn()` method outputs everything to the process standard output and error. To make it easier to test and debug, we can forward the output to the CLI Builder logger instead. This also allows the builder itself to be executed in a separate process, even if the standard output and error are deactivated (as in an Electron app).

We can retrieve a Logger instance from the context.

src/my-builder.ts (handling output)

```
import { BuilderContext, BuilderOutput, createBuilder } from '@angular-
devkit/architect';
import { JsonObject } from '@angular-devkit/core';
import * as childProcess from 'child_process';

interface Options extends JsonObject {
  command: string;
  args: string[];
}

export default createBuilder(commandBuilder);

function commandBuilder(
  options: Options,
  context: BuilderContext,
  ): Promise<BuilderOutput> {
    const child = childProcess.spawn(options.command, options.args);

    child.stdout.on('data', data => {
      context.logger.info(data.toString());
    });
    child.stderr.on('data', data => {
      context.logger.error(data.toString());
    });

    return new Promise(resolve => {
      child.on('close', code => {
        resolve({ success: code === 0 });
      });
    });
}
```

Progress and status reporting

The CLI Builder API includes progress and status reporting tools, which can provide hints for certain functions and interfaces.

To report progress, use the `BuilderContext.reportProgress()` method, which takes a current value, (optional) total, and status string as arguments. The total can be any number; for example, if you know how many files you have to process, the total could be the number of files, and current should be the number processed so far. The status string is unmodified unless you pass in a new string value.

You can see an example of how the `tslint` builder reports progress.

In our example, the shell command either finishes or is still executing, so there's no need for a progress report, but we can report status so that a parent builder that called our builder would know what's going on. Use the `BuilderContext.reportStatus()` method to generate a status string of any length. (Note that there's no guarantee that a long string will be shown entirely; it could be cut to fit the UI that displays it.) Pass an empty string to remove the status.

src/my-builder.ts (progess reporting)

```
import { BuilderContext, BuilderOutput, createBuilder } from '@angular-
devkit/architect';
import { JsonObject } from '@angular-devkit/core';
import * as childProcess from 'child_process';

interface Options extends JsonObject {
  command: string;
  args: string[];
}

export default createBuilder(commandBuilder);

function commandBuilder(
  options: Options,
  context: BuilderContext,
  ): Promise<BuilderOutput> {
    context.reportStatus(`Executing "${options.command}"...`);
    const child = childProcess.spawn(options.command, options.args);

    child.stdout.on('data', data => {
      context.logger.info(data.toString());
    });
    child.stderr.on('data', data => {
      context.logger.error(data.toString());
    });

    return new Promise(resolve => {
      context.reportStatus(`Done.`);
      child.on('close', code => {
        resolve({ success: code === 0 });
      });
    });
}
```

Builder input

You can invoke a builder indirectly through a CLI command, or directly with the Angular CLI `ng run` command. In either case, you must provide required inputs, but can allow other inputs to default to values that are pre-configured

for a specific *target*, provide a pre-defined, named override configuration, and provide further override option values on the command line.

Input validation

You define builder inputs in a JSON schema associated with that builder. The Architect tool collects the resolved input values into an `options` object, and validates their types against the schema before passing them to the builder function. (The Schematics library does the same kind of validation of user input).

For our example builder, we expect the `options` value to be a `JsonObject` with two keys: a `command` that is a string, and an `args` array of string values.

We can provide the following schema for type validation of these values.

command/schema.json

```
{
  "$schema": "http://json-schema.org/schema",
  "type": "object",
  "properties": {
    "command": {
      "type": "string"
    },
    "args": {
      "type": "array",
      "items": {
        "type": "string"
      }
    }
  }
}
```

> This is a very simple example, but the use of a schema for validation can be very powerful. For more information, see the JSON schemas website.

To link our builder implementation with its schema and name, we need to create a *builder definition* file, which we can point to in `package.json`.

Create a file named `builders.json` file that looks like this.

builders.json

```
{
  "builders": {
    "command": {
      "implementation": "./command",
```

```
      "schema": "./command/schema.json",
      "description": "Runs any command line in the operating system."
    }
  }
}
```

In the `package.json` file, add a `builders` key that tells the Architect tool where to find our builder definition file.

package.json

```
{
  "name": "@example/command-runner",
  "version": "1.0.0",
  "description": "Builder for Command Runner",
  "builders": "builders.json",
  "devDependencies": {
    "@angular-devkit/architect": "^1.0.0"
  }
}
```

The official name of our builder is now `@example/command-runner:command`. The first part of this is the package name (resolved using node resolution), and the second part is the builder name (resolved using the `builders.json` file).

Using one of our `options` is very straightforward, we did this in the previous section when we accessed `options.command`.

src/my-builder.ts (report status)

```
context.reportStatus(`Executing "${options.command}"...`);
const child = childProcess.spawn(options.command, options.args);
```

Target configuration

A builder must have a defined target that associates it with a specific input configuration and project.

Targets are defined in the `angular.json` CLI configuration file. A target specifies the builder to use, its default options configuration, and named alternative configurations. The Architect tool uses the target definition to resolve input options for a given run.

The `angular.json` file has a section for each project, and the "architect" section of each project configures targets for builders used by CLI commands such as 'build', 'test', and 'lint'. By default, for example, the `build` command runs the builder `@angular-devkit/build-angular:browser` to perform the build task, and passes in default option values as specified for the `build` target in `angular.json`.

angular.json

```
{
  "myApp": {
    ...
    "architect": {
      "build": {
        "builder": "@angular-devkit/build-angular:browser",
        "options": {
          "outputPath": "dist/myApp",
          "index": "src/index.html",
          ...
        },
        "configurations": {
          "production": {
            "fileReplacements": [
              {
                "replace": "src/environments/environment.ts",
                "with": "src/environments/environment.prod.ts"
              }
            ],
            "optimization": true,
            "outputHashing": "all",
            ...
          }
        }
      },
      ...
```

The command passes the builder the set of default options specified in the "options" section. If you pass the `--configuration=production` flag, it uses the override values specified in the `production` alternative configuration. You can specify further option overrides individually on the command line. You might also add more alternative configurations to the `build` target, to define other environments such as `stage` or `qa`.

Target strings

The generic `ng run` CLI command takes as its first argument a target string of the form *project:target[:configuration]*.

- *project*: The name of the Angular CLI project that the target is associated with.
- *target*: A named builder configuration from the `architect` section of the `angular.json` file.
- *configuration*: (optional) The name of a specific configuration override for the given target, as defined in the `angular.json` file.

If your builder calls another builder, it may need to read a passed target string. You can parse this string into an object by using the `targetFromTargetString()` utility function from `@angular-devkit/architect`.

Schedule and run

Architect runs builders asynchronously. To invoke a builder, you schedule a task to be run when all configuration resolution is complete.

The builder function is not executed until the scheduler returns a `BuilderRun` control object. The CLI typically schedules tasks by calling the `BuilderContext.scheduleTarget()` function, and then resolves input options using the target definition in the `angular.json` file.

Architect resolves input options for a given target by taking the default options object, then overwriting values from the configuration used (if any), then further overwriting values from the overrides object passed to `BuilderContext.scheduleTarget()`. For the Angular CLI, the overrides object is built from command line arguments.

Architect validates the resulting options values against the schema of the builder. If inputs are valid, Architect creates the context and executes the builder.

For more information see Workspace Configuration.

> You can also invoke a builder directly from another builder or test by calling `BuilderContext.scheduleBuilder()`. You pass an `options` object directly to the method, and those option values are validated against the schema of the builder without further adjustment.
>
> Only the `BuilderContext.scheduleTarget()` method resolves the configuration and overrides through the `angular.json` file.

Default architect configuration

Let's create a simple `angular.json` file that puts target configurations into context.

We can publish the builder to npm (see Publishing your Library), and install it using the following command:

```
npm install @example/command-runner
```

If we create a new project with `ng new builder-test`, the generated `angular.json` file looks something like this, with only default builder configurations.

angular.json

```
{
  // ...
  "projects": {
    // ...
    "builder-test": {
      // ...
      "architect": {
        // ...
        "build": {
```

```
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            // ... more options...
            "outputPath": "dist/builder-test",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "src/tsconfig.app.json"
          },
          "configurations": {
            "production": {
              // ... more options...
              "optimization": true,
              "aot": true,
              "buildOptimizer": true
            }
          }
        }
      }
    }
  }
  // ...
}
```

Adding a target

Let's add a new target that will run our builder to execute a particular command. This target will tell the builder to run `touch` on a file, in order to update its modified date.

We need to update the `angular.json` file to add a target for this builder to the "architect" section of our new project.

- We'll add a new target section to the "architect" object for our project.
- The target named "touch" uses our builder, which we published to `@example/command-runner`. (See Publishing your Library)
- The options object provides default values for the two inputs that we defined; `command`, which is the Unix command to execute, and `args`, an array that contains the file to operate on.
- The configurations key is optional, we'll leave it out for now.

angular.json

```
{
  "projects": {
    "builder-test": {
      "architect": {
        "touch": {
          "builder": "@example/command-runner:command",
```

```
          "options": {
            "command": "touch",
            "args": [
              "src/main.ts"
            ]
          }
        },
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/builder-test",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "src/tsconfig.app.json"
          },
          "configurations": {
            "production": {
              "fileReplacements": [
                {
                  "replace": "src/environments/environment.ts",
                  "with": "src/environments/environment.prod.ts"
                }
              ],
              "optimization": true,
              "aot": true,
              "buildOptimizer": true
            }
          }
        }
      }
    }
  }
}
```

Running the builder

To run our builder with the new target's default configuration, use the following CLI command in a Linux shell.

```
ng run builder-test:touch
```

This will run the `touch` command on the `src/main.ts` file.

You can use command-line arguments to override the configured defaults. For example, to run with a different `command` value, use the following CLI command.

```
ng run builder-test:touch --command=ls
```

This will call the `ls` command instead of the `touch` command. Because we did not override the *args* option, it will list information about the `src/main.ts` file (the default value provided for the target).

Testing a builder

Use integration testing for your builder, so that you can use the Architect scheduler to create a context, as in this example.

- In the builder source directory, we have created a new test file `my-builder.spec.ts`. The code creates new instances of `JsonSchemaRegistry` (for schema validation), `TestingArchitectHost` (an in-memory implementation of `ArchitectHost`), and `Architect`.
- We've added a `builders.json` file next to the builder's `package.json` file, and modified the package file to point to it.

Here's an example of a test that runs the command builder. The test uses the builder to run the `node --print 'foo'` command, then validates that the `logger` contains an entry for `foo`.

src/my-builder.spec.ts

```
import { Architect } from '@angular-devkit/architect';
import { TestingArchitectHost } from '@angular-devkit/architect/testing';
import { logging, schema } from '@angular-devkit/core';

describe('Command Runner Builder', () => {
  let architect: Architect;
  let architectHost: TestingArchitectHost;

  beforeEach(async () => {
    const registry = new schema.CoreSchemaRegistry();
    registry.addPostTransform(schema.transforms.addUndefinedDefaults);

    // TestingArchitectHost() takes workspace and current directories.
    // Since we don't use those, both are the same in this case.
    architectHost = new TestingArchitectHost(__dirname, __dirname);
    architect = new Architect(architectHost, registry);

    // This will either take a Node package name, or a path to the directory
    // for the package.json file.
    await architectHost.addBuilderFromPackage('..');
  });

  it('can run node', async () => {
    // Create a logger that keeps an array of all messages that were logged.
    const logger = new logging.Logger('');
    const logs = [];
```

```
    logger.subscribe(ev => logs.push(ev.message));

    // A "run" can have multiple outputs, and contains progress information.
    const run = await architect.scheduleBuilder('@example/command-runner:command', {
      command: 'node',
      args: ['--print', '\'foo\''],
    }, { logger });  // We pass the logger for checking later.

    // The "result" member (of type BuilderOutput) is the next output.
    const output = await run.result;

    // Stop the builder from running. This stops Architect from keeping
    // the builder-associated states in memory, since builders keep waiting
    // to be scheduled.
    await run.stop();

    // Expect that foo was logged
    expect(logs).toContain('foo');
  });
});
```

> When running this test in your repo, you need the `ts-node` package. You can avoid this by renaming `my-builder.spec.ts` to `my-builder.spec.js`.

Watch mode

Architect expects builders to run once (by default) and return. This behavior is not entirely compatible with a builder that watches for changes (like Webpack, for example). Architect can support watch mode, but there are some things to look out for.

- To be used with watch mode, a builder handler function should return an Observable. Architect subscribes to the Observable until it completes and might reuse it if the builder is scheduled again with the same arguments.
- The builder should always emit a `BuilderOutput` object after each execution. Once it's been executed, it can enter a watch mode, to be triggered by an external event. If an event triggers it to restart, the builder should execute the `BuilderContext.reportRunning()` function to tell Architect that it is running again. This prevents Architect from stopping the builder if another run is scheduled.

When your builder calls `BuilderRun.stop()` to exit watch mode, Architect unsubscribes from the builder's Observable and calls the builder's teardown logic to clean up. (This behavior also allows for long running builds to be stopped and cleaned up.)

In general, if your builder is watching an external event, you should separate your run into three phases.

1. **Running** For example, webpack compiles. This ends when webpack finishes and your builder emits a `BuilderOutput` object.

2. **Watching** Between two runs, watch an external event stream. For example, webpack watches the file system for any changes. This ends when webpack restarts building, and `BuilderContext.reportRunning()` is called. This goes back to step 1.

3. **Completion** Either the task is fully completed (for example, webpack was supposed to run a number of times), or the builder run was stopped (using `BuilderRun.stop()`). Your teardown logic is executed, and Architect unsubscribes from your builder's Observable.

Summary

The CLI Builder API provides a new way of changing the behavior of the Angular CLI by using builders to execute custom logic.

- Builders can be synchronous or asynchronous, execute once or watch for external events, and can schedule other builders or targets.
- Builders have option defaults specified in the `angular.json` configuration file, which can be overwritten by an alternate configuration for the target, and further overwritten by command line flags.
- We recommend that you use integration tests to test Architect builders. You can use unit tests to validate the logic that the builder executes.
- If your builder returns an Observable, it should clean up in the teardown logic of that Observable.

Version 10.0.10-local+sha.84d1ba792b.