# Compliance test-cases

This directory contains rules, helpers and test-cases for the Angular compiler compliance tests.

There are three different types of tests that are run based on file-based "test-cases".

- **Full compile** - in this test the source files defined by the test-case are fully compiled by Angular. The generated files are compared to "expected files" via a matching algorithm that is tolerant to whitespace and variable name changes.
- **Partial compile** - in this test the source files defined by the test-case are "partially" compiled by Angular to produce files that can be published. These partially compiled files are compared directly against "golden files" to ensure that we do not inadvertently break the public API of partial declarations.
- **Linked** - in this test the golden files mentioned in the previous bullet point, are passed to the Angular linker, which generates files that are comparable to the fully compiled files. These linked files are compared against the "expected files" in the same way as in the "full compile" tests.

This way the compliance tests are able to check each mode and stage of compilation is accurate and does not change unexpectedly.

## Defining a test-case

To define a test-case, create a new directory below `test_cases`. In this directory

- add a new file called `TEST_CASES.json`. The format of this file is described below.
- add an empty `GOLDEN_PARTIAL.js` file. This file will be updated by the tooling later.
- add any `inputFiles` that will be compiled as part of the test-case.
- add any `expected` files that will be compared to the files generated by compiling the source files.

### TEST_CASES.json format

The `TEST_CASES.json` defines an object with one or more test-case definitions in the `cases` property.

Each test-case can specify:

- A `description` of the test.
- The `inputFiles` that will be compiled.
- Additional `compilerOptions` and `angularCompilerOptions` that are passed to the compiler.

- Whether to exclude this test-case from certain tests running under certain compilation modes (`compilationModeFilter`).
- A collection of `expectations` definitions that will be checked against the generated files.

Note that there is a JSON schema for the `TEST_CASES.json` file stored at `test_cases/test_case_schema.json`. You should add a link to this schema at the top of your `TEST_CASES.json` file to provide validation and intellisense for this file in your IDE.

For example:

```
{
  "$schema": "../test_case_schema.json",
  "cases": [
    {
      "description": "description of the test - equivalent to an `it` clause message.",
      "inputFiles": ["abc.ts"],
      "expectations": [
        {
          "failureMessage": "message to display if this expectation fails",
          "files": [
            { "expected": "xyz.js", "generated": "abc.js" }, ...
          ]
        }, ...
      ],
      "compilerOptions": { ... },
      "angularCompilerOptions": { ... }
    }
  ]
}
```

### Input files

The input files are the source file that will be compiled as part of this test-case. Input files should be stored in the directory next to the `TEST_CASES.json`. The paths to the input files should be listed in the `inputFiles` property of `TEST_CASES.json`. The paths are relative to the `TEST_CASES.json` file.

If no `inputFiles` property is provided, the default is `["test.ts"]`.

Note that test-cases can share input files, but you should only do this if these input files are going to be compiled using the same options. This is because only one version of the compiled input file is retrieved from the golden partial file to be used in the linker tests. This can cause the linker tests to fail if they are provided with a compiled file (from the golden partial) that was compiled with different options to what are expected for that test-case.

### Expectations

An expectation consists of a `failureMessage`, which is displayed if the expectation check fails, a collection of expected `files` pairs and/or a collection of `expectedErrors`.

Each expected file-pair consists of a path to a `generated` file (relative to the build output folder), and a path to an `expected` file (relative to the test case).

The `generated` file is checked to see if it "matches" the `expected` file. The matching is resilient to whitespace and variable name changes.

If no `files` property is provided, the default is a a collection of objects `{expected, generated}`, where `expected` and `generated` are computed by taking each path in the `inputFiles` collection and replacing the `.ts` extension with `.js`.

Each expected error must have a `message` property and, optionally, a `location` property. These are parsed as regular expressions (so `.` and `(` etc must be escaped) and tested against the errors that are returned as diagnostics from the compilation.

If no `failureMessage` property is provided, the default is `"Incorrect generated output."`.

### Expected file format

The expected files look like JavaScript but are actually specially formatted to allow matching with the generated output. The generated and expected files are tokenized and then the tokens are intelligently matched to check whether they are equivalent.

- Whitespace tolerant - the tokens can be separated by any amount of whitespace
- Code skipping - you can skip sections of code in the generated output by adding an ellipsis (`...`) to the expectation file.
- Identifier tolerant - identifiers in the expectation file that start and end with a dollar (e.g. `$r3$`) will be matched against any identifier. But the matching will ensure that the same identifier name appears consistently elsewhere in the file.
- Macro expansion - we can add macros to the expected files that will be expanded to blocks of code dynamically. The following macros are defined in the `test_helpers/expected_file_macros.ts` file:
  - I18n messages - for example: `__i18nMsg__('message string', [ ['placeholder', 'pair] ], { meta: 'properties'})`.
  - Attribute markers - for example: `__AttributeMarker.Bindings__`.

#### Source-map checks

To check a mapping, add a `// SOURCE:` comment to the end of a line in an expectation file:

```
<generated code> // SOURCE: "<source-url>" "<source code>"
```

The generated code, stripped of the `// SOURCE:` comment, will still be checked as normal by the `expectEmit()` helper. But, prior to that, the source-map segments are checked to ensure that there is a mapping from `<generated code>` to `<source code>` found in the file at `<source-url>`.

Note:

- The source-url should be absolute, with the directory containing the TEST_CASES.json file assumed to be `/`.
- Whitespace is important and will be included when comparing the segments.
- There is a single space character between each part of the line.
- Double quotes in the mapping must be escaped.
- Newlines within a mapping must be escaped since the mapping and comment must all appear on a single line of this file.

## Running tests

The simplest way to run all the compliance tests is:

```
yarn test //packages/compiler-cli/test/compliance/...
```

If you only want to run one of the three types of test you can be more specific:

```
yarn test //packages/compiler-cli/test/compliance/full
yarn test //packages/compiler-cli/test/compliance/linked
yarn test //packages/compiler-cli/test/compliance/test_cases/...
```

(The last command runs the partial compilation tests.)

## Updating a golden partial file

There is one golden partial file per `TEST_CASES.json` file. So even if this file defines multiple test-cases, which each contain multiple input files, there will only be one golden file.

The golden file is generated by the tooling and should not be modified manually.

When you first create a test-case, with an empty `GOLDEN_PARTIAL.js` file, or a change is made to the generated partial output, we must update the `GOLDEN_PARTIAL.js` file.

This is done by running a specific bazel rule of the form:

```
bazel run //packages/compiler-cli/test/compliance/test_cases:<path/to/test_case>.golden.upda
```

where to replace `<path/to/test_case>` with the path (relative to `test_cases`) of the directory that contains the `GOLDEN_PARTIAL.js` to update.

To update all golden partial files, the following command can be run:

```
node packages/compiler-cli/test/compliance/update_all_goldens.js
```

## Debugging test-cases

The full and linked compliance tests are basically `jasmine_node_test` rules. As such, they can be debugged just like any other `jasmine_node_test`. The standard approach is to add `--config=debug` to the Bazel test command.

For example:

```
yarn test //packages/compiler-cli/test/compliance/full --config=debug
yarn test //packages/compiler-cli/test/compliance/linked --config=debug
```

To debug generating the partial golden output use the following form of Bazel command:

```
yarn bazel run //packages/compiler-cli/test/compliance/test_cases:generate_partial_for_<path
```

The `path/to/test_case` is relative to the `test_cases` directory. So for this `TEST_CASES.json` file at:

```
packages/compiler-cli/test/compliance/test_cases/r3_view_compiler_directives/directives/matc
```

The command to debug the test-cases would be:

```
yarn bazel run //packages/compiler-cli/test/compliance/test_cases:generate_partial_for_r3_vi
```

### Focusing test-cases

You can focus a test case by setting `"focusTest": true` in the `TEST_CASES.json` file. This is equivalent to using jasmine `fit()`.

### Excluding test-cases

You can exclude a test case by setting `"excludeTest": true` in the `TEST_CASES.json` file. This is equivalent to using jasmine `xit()`.