Integration tests for Angular

This directory contains end-to-end tests for Angular. Each directory is a self-contained application that exactly mimics how a user might expect Angular to work, so they allow high-fidelity reproductions of real-world issues.

For this to work, we first build the Angular distribution via `./scripts/build/build-packages-dist.js`, then install the distribution into each app.

To test Angular CLI applications, we use the `cli-hello-world-*` integration tests. When a significant change is released in the CLI, the applications should be updated with `ng update`:

```
$ cd integration/cli-hello-world[-*]
$ yarn install
$ yarn ng update @angular/cli @angular-devkit/build-angular
$ yarn build
$ yarn test
```

Afterwards the `@angular/cli` and `@angular-devkit/build-angular` should be reverted to the `file:../` urls and the main `package.json` should be updated with the new versions.

Render3 tests

The directory `cli-hello-world-ivy-compat` contains a test for render3 used with the angular cli.

The `cli-hello-world-ivy-minimal` contains a minimal ivy app that is meant to mimic the bazel equivalent in `packages/core/test/bundling/hello_world`, and should be kept similar.

Writing an integration test

The API for each test is:

- Each sub-directory here is an integration test
- Each test should have a `package.json` file
- The test runner will run `yarn` and `yarn test` on the package

This means that the test should be started by test script, like

```
"scripts": {"test": "runProgramA && assertResultIsGood"}
```

Note that the `package.json` file uses a special `file:../../dist` scheme to reference the Angular packages, so that the locally-built Angular is installed into the test app.

Also, beware of floating (non-locked) dependencies. If in doubt, you can install the package directly from `file:../../node_modules`.

WARNING

Always ensure that `yarn.lock` files are up-to-date with the corresponding `package.json` files (wrt the non-local dependencies - i.e. dependencies whose versions do not start with `file:`).

You can update a `yarn.lock` file by running `yarn install` in the project subdirectory.

Running integration tests

```
$ ./integration/run_tests.sh
```

The test runner will first re-build any stale npm packages, then `cd` into each subdirectory to execute the test.

Running integration tests under Bazel

The PR https://github.com/angular/angular/pull/33927 added the ability to run integration tests with Bazel. These tests can be resource intensive so it is recommended to limit the number of concurrent test jobs with the `--local_test_jobs` bazel flag.

Locally, if Bazel uses all of your cores to run the maximum number of integration tests in parallel then this can lead to test timeouts and flakes and freeze up your machine while these tests are running. You can limit the number of concurrent local integration tests that run with:

```
yarn bazel test --local_test_jobs=<N> //integration/...
```

Set a reasonable `local_test_jobs` limit for your local machine to prevent full cpu utilization during local development test runs.

To avoid having to specify this command line flag, you may want to include it in your `.bazelrc.user` file:

```
test --local_test_jobs=<N>
```

The downside of this is that this will apply to all tests and not just the resource intensive integration tests.

Bazel-in-bazel integration tests

Two of the integration tests that run Bazel-in-Bazel are particularly resource intensive and are tagged “manual” and “exclusive”. To run these tests use,

```
yarn bazel test //integration/bazel:test
yarn bazel test //integration/cli-hello-world-ivy-minimal:test
```

Adding a new integration test

When adding a new integration test, follow the steps below to add a bazel test target for the new test.

1. Add a build file using the `ng_integration_test` rule from `//integration:index.bzl`.
2. If test requires ports and does not support ethereal ports then make sure the port is unique and add it to the "manually configured ports" section to document which port it is using
3. Add at least the following two entries `.bazelignore` (as they may contain BUILD files)
    1. `integration/new_test/node_modules`
    2. `integration/new_test/.yarn_local_cache`
4. Add any other untracked folders to `.bazelignore` that may contain `BUILD` files
5. If there are BUILD files in the integration test folder (except for the top-level one defining the test), add those folders to the `--deleted_packages` in the `.bazelrc`. An example is the `bazel_ngtsc_plugin` test within `//integration/bazel_workspace_tests`.

Manually configured ports

Some integration ports must be managed manually to be unique and in other cases the tests are able to select a random free port.

Where `ng e2e` is used we pass `ng e2e --port 0` which prompts the cli to select a random free port for the e2e test. The protractor.conf is automatically updated to use this port.

Karma automatically finds a free port so no effort is needed there.

The manually configured ports are as follows:

TEST	PORT	CONFIGURATION
dynamic-compiler	4201	/e2e/browser.config.json: "port": 4201
hello_world___closure	4202	/e2e/browser.config.json: "port": 4202
i18n	4204	/e2e/browser.config.json: "port": 4204
ng_elements	4205	/e2e/browser.config.json: "port": 4205
platform-server	4206	/src/server.ts: app.listen(4206,...

Note: This will become obsolete soon once we start running integration tests with RBE and within a sandbox environment.

Browser tests

For integration tests we use the Bazel-managed versions of `chromium`. For both Karma and Protractor tests we set a number of browser testing flags. To avoid duplication, they will be listed and explained here and the code will reference this file for more information.

No Sandbox: –no-sandbox

The sandbox needs to be disabled with the `--no-sandbox` flag for both Karma and Protractor tests, because it causes Chrome to crash on some environments.

See: https://chromedriver.chromium.org/help/chrome-doesn-t-start See: https://github.com/puppeteer/puppeteer/blob/v1.0.0/docs/troubleshooting.md#chrome-headless-fails-due-to-sandbox-issues

Headless: –headless

So that browsers are not popping up and tearing down when running these tests we run Chrome in headless mode. The `--headless` flag puts Chrome in headless mode and a number of other flags are recommended in this mode as well:

- `--headless`
- `--disable-gpu`
- `--disable-dev-shm-usage`
- `--hide-scrollbars`
- `--mute-audio`

These come from the flags that puppeteer passes to chrome when it launches it in headless mode: https://github.com/puppeteer/puppeteer/blob/18f2ecdffdfc70e891750b570bfe8bea5b5ca8c2/lib/

And from the flags that the Karma `ChromeHeadless` browser passes to Chrome: https://github.com/karma-runner/karma-chrome-launcher/blob/5f70a76de87ecbb57f3f3cb556aa6a2a1a4f643f/in

Disable shared memory space: –disable-dev-shm-usage The `--disable-dev-shm-usage` flag disables the usage of `/dev/shm` because it causes Chrome to crash on some environments.

On CircleCI, the puppeteer provisioned Chrome crashes with `CI we get Root cause: org.openqa.selenium.WebDriverException: unknown error: DevToolsActivePort file doesn't exist which resolves` without this flag.

See: https://github.com/puppeteer/puppeteer/blob/v1.0.0/docs/troubleshooting.md#tips See: https://stackoverflow.com/questions/50642308/webdriverexception-unknown-error-devtoolsactiveport-file-doesnt-exist-while-t

Debugging Size Regressions

If size regression occurs, one way to debug is to get a build which shows the code before and after. Here are the steps to do that.

1. Check out both the `master` branch as well as your change (let's refer to it as `change` branch) into two different working locations. (A suggested way to do this is using `git worktree`.)
2. In both `master` and `change` locations update the failing tests `package.json` with `NG_BUILD_DEBUG_OPTIMIZE=minify` environment variable so that the resulting build would contain a human readable but optimized output. As an example:
   - Open `integration/cli-hello-world/package.json` and prefix `NG_BUILD_DEBUG_OPTIMIZE=minify` into the `build` rule. Resulting in something like: `"build": "NG_BUILD_DEBUG_OPTIMIZE=minify ng build --prod",`
   - Run `bazel test //integration/cli-hello-world:test --test_output=streamed --cache_test_results=no` to run the test.
   - Open the test temporary directory as printed out by Bazel.
   - Diff the `master` vs `change` to see the differences. `myDiffTool change/integration/cli-hello-world/dist/main-es2015.*.js master/integration/cli-hello-world/dist/main-es2015.*.js`
   - The above should give you a better understanding as to what has changed and what is causing the regression.