{@a top}

Testing

Testing your Angular application helps you check that your app is working as you expect.

Prerequisites

Before writing tests for your Angular app, you should have a basic understanding of the following concepts:

- Angular fundamentals
- JavaScript
- HTML
- CSS
- Angular CLI

The testing documentation offers tips and techniques for unit and integration testing Angular applications through a sample application created with the Angular CLI. This sample application is much like the one in the *Tour of Heroes* tutorial.

If you'd like to experiment with the application that this guide describes, run it in your browser or download and run it locally.

{@a setup}

Set up testing

The Angular CLI downloads and installs everything you need to test an Angular application with the Jasmine test framework.

The project you create with the CLI is immediately ready to test. Just run the `ng test` CLI command:

ng test
The `ng test` command builds the application in *watch mode*, and launches the Karma test runner.

The console output looks a bit like this:

10% building modules 1/1 modules 0 active ...INFO [karma]: Karma v1.7.1 server started at http://0.0.0.0:9876/ ...INFO [launcher]: Launching browser Chrome ... ...INFO [launcher]: Starting browser Chrome ...INFO [Chrome ...]: Connected on socket ... Chrome ...: Executed 3 of 3 SUCCESS (0.135 secs / 0.205 secs)
The last line of the log is the most important. It shows that Karma ran three tests that all passed.

A Chrome browser also opens and displays the test output in the "Jasmine HTML Reporter" like this.

 Jasmine HTML Reporter in the browser

Most people find this browser output easier to read than the console log. Click on a test row to re-run just that test or click on a description to re-run the tests in the selected test group ("test suite").

Meanwhile, the `ng test` command is watching for changes.

To see this in action, make a small change to `app.component.ts` and save. The tests run again, the browser refreshes, and the new test results appear.

Configuration

The CLI takes care of Jasmine and Karma configuration for you.

Fine-tune many options by editing the `karma.conf.js` in the root folder of the project and the `test.ts` files in the `src/` folder.

The `karma.conf.js` file is a partial Karma configuration file. The CLI constructs the full runtime configuration in memory, based on application structure specified in the `angular.json` file, supplemented by `karma.conf.js`.

Search the web for more details about Jasmine and Karma configuration.

Other test frameworks

You can also unit test an Angular application with other testing libraries and test runners. Each library and runner has its own distinctive installation procedures, configuration, and syntax.

Search the web to learn more.

Test file name and location

Look inside the `src/app` folder.

The CLI generated a test file for the `AppComponent` named `app.component.spec.ts`.

The test file extension **must be** **`.spec.ts`** so that tooling can identify it as a file with tests (also known as a *spec* file).

The `app.component.ts` and `app.component.spec.ts` files are siblings in the same folder. The root file names (`app.component`) are the same for both files.

Adopt these two conventions in your own projects for *every kind* of test file.

{@a q-spec-file-location}

Place your spec file next to the file it tests

It's a good idea to put unit test spec files in the same folder as the application source code files that they test:

- Such tests are painless to find.
- You see at a glance if a part of your application lacks tests.
- Nearby tests can reveal how a part works in context.
- When you move the source (inevitable), you remember to move the test.
- When you rename the source file (inevitable), you remember to rename the test file.

{@a q-specs-in-test-folder}

Place your spec files in a test folder

Application integration specs can test the interactions of multiple parts spread across folders and modules. They don't really belong to any part in particular, so they don't have a natural home next to any one file.

It's often better to create an appropriate folder for them in the `tests` directory.

Of course specs that test the test helpers belong in the `test` folder, next to their corresponding helper files.

{@a ci}

Set up continuous integration

One of the best ways to keep your project bug-free is through a test suite, but you might forget to run tests all the time. Continuous integration (CI) servers let you set up your project repository so that your tests run on every commit and pull request.

There are paid CI services like Circle CI and Travis CI, and you can also host your own for free using Jenkins and others. Although Circle CI and Travis CI are paid services, they are provided free for open source projects. You can create a public project on GitHub and add these services without paying. Contributions to the Angular repository are automatically run through a whole suite of Circle CI tests.

This article explains how to configure your project to run Circle CI and Travis CI, and also update your test configuration to be able to run tests in the Chrome browser in either environment.

Configure project for Circle CI

Step 1: Create a folder called `.circleci` at the project root.

Step 2: In the new folder, create a file called `config.yml` with the following content:

```
version: 2
jobs:
  build:
    working_directory: ~/my-project
    docker:
      - image: circleci/node:10-browsers
    steps:
      - checkout
      - restore_cache:
          key: my-project-{{ .Branch }}-{{ checksum "package-lock.json" }}
      - run: npm install
      - save_cache:
          key: my-project-{{ .Branch }}-{{ checksum "package-lock.json" }}
          paths:
            - "node_modules"
      - run: npm run test -- --no-watch --no-progress --browsers=ChromeHeadlessCI
```

This configuration caches `node_modules/` and uses `npm run` to run CLI commands, because `@angular/cli` is not installed globally. The double dash (`--`) is needed to pass arguments into the `npm` script.

Step 3: Commit your changes and push them to your repository.

Step 4: Sign up for Circle CI and add your project. Your project should start building.

- Learn more about Circle CI from Circle CI documentation.

Configure project for Travis CI

Step 1: Create a file called `.travis.yml` at the project root, with the following content:

```
language: node_js
node_js:
  - "10"
```

```
addons:
  chrome: stable

cache:
  directories:
    - ./node_modules

install:
  - npm install

script:
  - npm run test -- --no-watch --no-progress --browsers=ChromeHeadlessCI
```

This does the same things as the CircleCI configuration, except that Travis doesn't come with Chrome, so use Chromium instead.

Step 2: Commit your changes and push them to your repository.

Step 3: Sign up for Travis CI and add your project. You'll need to push a new commit to trigger a build.

- Learn more about Travis CI testing from Travis CI documentation.

Configure project for GitLab CI

Step 1: Create a file called `.gitlab-ci.yml` at the project root, with the following content:

```
image: node:14.15-stretch
variables:
  FF_USE_FASTZIP: "true"

cache:
  untracked: true
  policy: push
  key: ${CI_COMMIT_SHORT_SHA}
  paths:
    - node_modules/

.pull_cached_node_modules:
  cache:
    untracked: true
    key: ${CI_COMMIT_SHORT_SHA}
    policy: pull

stages:
  - setup
  - test

install:
  stage: setup
  script:
    - npm ci

test:
```

```
  stage: test
  extends: .pull_cached_node_modules
  before_script:
    - apt-get update
    - wget https://dl.google.com/linux/direct/google-chrome-stable_current_amd64.deb
    - apt install -y ./google-chrome*.deb;
    - export CHROME_BIN=/usr/bin/google-chrome
  script:
    - npm run test -- --no-watch --no-progress --browsers=ChromeHeadlessCI
```

This configuration caches `node_modules/` in the `install` job and re-uses the cached `node_modules/` in the `test` job.

Step 2: Sign up for GitLab CI and add your project. You'll need to push a new commit to trigger a build.

Step 3: Commit your changes and push them to your repository.

- Learn more about GitLab CI testing from GitLab CI/CD documentation.

Configure project for GitHub Actions

Step 1: Create a folder called `.github/workflows` at root of your project

Step 2: In the new folder, create a file called `main.yml` with the following content:

```
name: CI Angular app through Github Actions
on: push
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - name: Use Node.js 14.x
        uses: actions/setup-node@v1
        with:
          node-version: 14.x

      - name: Setup
        run: npm ci

      - name: Test
        run: |
          npm test -- --no-watch --no-progress --browsers=ChromeHeadlessCI
```

Step 3: Sign up for GitHub and add your project. You'll need to push a new commit to trigger a build.

Step 4: Commit your changes and push them to your repository.

- Learn more about GitHub Actions from GitHub Actions documentation.

Configure CLI for CI testing in Chrome

While the CLI command `ng test` is generally running the CI tests in your environment, you might still need to adjust your configuration to run the Chrome browser tests.

There is a configuration file for the Karma JavaScript test runner, which you must adjust to start Chrome without sandboxing.

We'll be using Headless Chrome in these examples.

- In the Karma configuration file, `karma.conf.js`, add a custom launcher called ChromeHeadlessCI below browsers:

```
browsers: ['ChromeHeadlessCI'],
customLaunchers: {
  ChromeHeadlessCI: {
    base: 'ChromeHeadless',
    flags: ['--no-sandbox']
  }
},
```

Now, run the following command to use the `--no-sandbox` flag:

ng test --no-watch --no-progress --browsers=ChromeHeadlessCI

Note: Right now, you'll also want to include the `--disable-gpu` flag if you're running on Windows. See crbug.com/737678.

More information on testing

After you've set up your application for testing, you might find the following testing guides useful.

- Code coverage—find out how much of your app your tests are covering and how to specify required amounts.
- Testing services—learn how to test the services your application uses.
- Basics of testing components—discover the basics of testing Angular components.
- Component testing scenarios—read about the various kinds of component testing scenarios and use cases.
- Testing attribute directives—learn about how to test your attribute directives.
- Testing pipes—find out how to test pipes.
- Debugging tests—uncover common testing bugs.
- Testing utility APIs—get familiar with Angular testing features.