Developing AngularJS

Development Setup

This document describes how to set up your development environment to build and test AngularJS, and explains the basic mechanics of using `git`, `node`, `yarn` and `grunt`.

Installing Dependencies

Before you can build AngularJS, you must install and configure the following dependencies on your machine:

- Git: The Github Guide to Installing Git is a good source of information.
- Node.js v8.x (LTS): We use Node to generate the documentation, run a development web server, run tests, and generate distributable files. Depending on your system, you can install Node either from source or as a pre-packaged bundle.

  We recommend using nvm (or nvm-windows) to manage and install Node.js, which makes it easy to change the version of Node.js per project.
- Yarn: We use Yarn to install our Node.js module dependencies (rather than using npm). See the detailed installation instructions.
- Java: We minify JavaScript using Closure Tools, which require Java (version 7 or higher) to be installed and included in your PATH variable.
- Grunt: We use Grunt as our build system. We're using it as a local dependency, but you can also add the grunt command-line tool globally (with `yarn global add grunt-cli`), which allows you to leave out the `yarn` prefix for all our grunt commands.

Forking AngularJS on Github

To contribute code to AngularJS, you must have a GitHub account so you can push code to your own fork of AngularJS and open Pull Requests in the GitHub Repository.

To create a Github account, follow the instructions here. Afterwards, go ahead and fork the main AngularJS repository.

Building AngularJS

To build AngularJS, you clone the source code repository and use Grunt to generate the non-minified and minified AngularJS files:

```
# Clone your Github repository:
git clone https://github.com/<github username>/angular.js.git

# Go to the AngularJS directory:
cd angular.js
```

```
# Add the main AngularJS repository as an upstream remote to your repository:
git remote add upstream "https://github.com/angular/angular.js.git"

# Install JavaScript dependencies:
yarn install

# Build AngularJS:
yarn grunt package
```

Note: If you're using Windows, you must use an elevated command prompt (right click, run as Administrator). This is because `yarn grunt package` creates some symbolic links.

The build output is in the `build` directory. It consists of the following files and directories:

- `angular-<version>.zip` — The complete zip file, containing all of the release build artifacts.
- `angular.js` / `angular.min.js` — The regular and minified core AngularJS script file.
- `angular-*.js` / `angular-*.min.js` — All other AngularJS module script files.
- `docs/` — A directory that contains a standalone version of the docs (same as served in `docs.angularjs.org`).

Running a Local Development Web Server

To debug code, run end-to-end tests, and serve the docs, it is often useful to have a local HTTP server. For this purpose, we have made available a local web server based on Node.js.

1. To start the web server, run:

   ```
   yarn grunt webserver
   ```

2. To access the local server, enter the following URL into your web browser:

   ```
   http://localhost:8000/
   ```

   By default, it serves the contents of the AngularJS project directory.

3. To access the locally served docs, visit this URL:

   ```
   http://localhost:8000/build/docs/
   ```

Running Tests

Running the Unit Test Suite

We write unit and integration tests with Jasmine and execute them with Karma. To run all of the tests once on Chrome run:

```
yarn grunt test:unit
```

To run the tests on other browsers use the command line flag:

```
yarn grunt test:unit --browsers=Chrome,Firefox
```

Note: there should be *no spaces between browsers*. `Chrome, Firefox` is INVALID.

If you have a Saucelabs or Browserstack account, you can also run the unit tests on these services via our pre-defined customLaunchers. See the karma config file for all pre-configured browsers.

For example, to run the whole unit test suite on selected browsers:

```
# Browserstack
yarn grunt test:unit --
browsers=BS_Chrome,BS_Firefox,BS_Safari,BS_IE_9,BS_IE_10,BS_IE_11,BS_EDGE,BS_iOS_10
# Saucelabs
yarn grunt test:unit --
browsers=SL_Chrome,SL_Firefox,SL_Safari,SL_IE_9,SL_IE_10,SL_IE_11,SL_EDGE,SL_iOS_10
```

Running these commands requires you to set up Karma Browserstack or Karma-Saucelabs, respectively.

During development, however, it's more productive to continuously run unit tests every time the source or test files change. To execute tests in this mode run:

1. To start the Karma server, capture Chrome browser and run unit tests, run:

   ```
   yarn grunt autotest
   ```

2. To capture more browsers, open this URL in the desired browser (URL might be different if you have multiple instance of Karma running, read Karma's console output for the correct URL):

   ```
   http://localhost:9876/
   ```

3. To re-run tests just change any source or test file.

To learn more about all of the preconfigured Grunt tasks run:

```
yarn grunt --help
```

Running the End-to-end Test Suite

AngularJS's end to end tests are run with Protractor. Simply run:

```
yarn grunt test:e2e
```

This will start the webserver and run the tests on Chrome.

Coding Rules

To ensure consistency throughout the source code, keep these rules in mind as you are working:

- All features or bug fixes **must be tested** by one or more specs.
- All public API methods **must be documented** with ngdoc, an extended version of jsdoc (we added support for markdown and templating via @ngdoc tag). To see how we document our APIs, please check out the existing source code and see the section about writing documentation
- With the exceptions listed below, we follow the rules contained in Google's JavaScript Style Guide:
  - **Do not use namespaces**: Instead, wrap the entire AngularJS code base in an anonymous closure and export our API explicitly rather than implicitly.
  - Wrap all code at **100 characters**.
  - Instead of complex inheritance hierarchies, we **prefer simple objects**. We use prototypal inheritance only when absolutely necessary.
  - We **love functions and closures** and, whenever possible, prefer them over objects.
  - To write concise code that can be better minified, we **use aliases internally** that map to the external API. See our existing code to see what we mean.
  - We **don't go crazy with type annotations** for private internal APIs unless it's an internal API that is used throughout AngularJS. The best guidance is to do what makes the most sense.

Specific topics

Provider configuration

When adding configuration (options) to providers, we follow a special pattern.

- for each option, add a `method` that ...
  - works as a getter and returns the current value when called without argument
  - works as a setter and returns itself for chaining when called with argument
  - for boolean options, uses the naming scheme `<option>Enabled([enabled])`
- non-primitive options (e.g. objects) should be copied or the properties assigned explicitly to a new object so that the configuration cannot be changed during runtime.

For a boolean config example, see `$compileProvider#debugInfoEnabled`

For an object config example, see `$location.html5Mode`

Throwing errors

User-facing errors should be thrown with `minErr` , a special error function that provides errors ids, templated error messages, and adds a link to a detailed error description.

The `$compile:badrestrict` error is a good example for a well-defined `minErr` : code and description.

Git Commit Guidelines

We have very precise rules over how our git commit messages can be formatted. This leads to **more readable messages** that are easy to follow when looking through the **project history**. But also, we use the git commit messages to **generate the AngularJS change log**.

The commit message formatting can be added using a typical git workflow or through the use of a CLI wizard (Commitizen). To use the wizard, run `yarn run commit` in your terminal after staging your changes in git.

Commit Message Format

Each commit message consists of a **header**, a **body** and a **footer**. The header has a special format that includes a **type**, a **scope** and a **subject**:

```
<type>(<scope>): <subject>
<BLANK LINE>
<body>
<BLANK LINE>
<footer>
```

The **header** is mandatory and the **scope** of the header is optional.

Any line of the commit message cannot be longer than 100 characters! This allows the message to be easier to read on GitHub as well as in various git tools.

Revert

If the commit reverts a previous commit, it should begin with `revert:` , followed by the header of the reverted commit. In the body it should say: `This reverts commit <hash>.` , where the hash is the SHA of the commit being reverted.

Type

Must be one of the following:

- **feat**: A new feature
- **fix**: A bug fix
- **docs**: Documentation only changes
- **style**: Changes that do not affect the meaning of the code (white-space, formatting, missing semi-colons, etc)
- **refactor**: A code change that neither fixes a bug nor adds a feature
- **perf**: A code change that improves performance
- **test**: Adding missing or correcting existing tests
- **chore**: Changes to the build process or auxiliary tools and libraries such as documentation generation

Scope

The scope could be anything specifying place of the commit change. For example `$location` , `$browser` , `$compile` , `$rootScope` , `ngHref` , `ngClick` , `ngView` , etc...

You can use `*` when the change affects more than a single scope.

Subject

The subject contains succinct description of the change:

- use the imperative, present tense: "change" not "changed" nor "changes"
- don't capitalize first letter
- no dot (.) at the end

Body

Just as in the **subject**, use the imperative, present tense: "change" not "changed" nor "changes". The body should include the motivation for the change and contrast this with previous behavior.

Footer

The footer should contain any information about **Breaking Changes** and is also the place to reference GitHub issues that this commit closes.

Breaking Changes should start with the word `BREAKING CHANGE:` with a space or two newlines. The rest of the commit message is then used for this.

A detailed explanation can be found in this document.

Writing Documentation

The AngularJS project uses a form of jsdoc called ngdoc for all of its code documentation.

This means that all the docs are stored inline in the source code and so are kept in sync as it changes.

There is also extra content (the developer guide, error pages, the tutorial, and misceallenous pages) that live inside the AngularJS repository as markdown files.

This means that since we generate the documentation from the source code, we can easily provide version-specific documentation by simply checking out a version of AngularJS and running the build.

Extracting the source code documentation, processing and building the docs is handled by the documentation generation tool Dgeni.

Building and viewing the docs locally

The docs can be built from scratch using grunt:

```
yarn grunt docs
```

This defers the doc-building task to `gulp`.

Note that the docs app is using the local build files to run. This means you might first have to run the build:

```
yarn grunt build
```

(This is also necessary if you are making changes to minErrors).

To view the docs, see Running a Local Development Web Server.

Writing jsdoc

The ngdoc utility has basic support for many of the standard jsdoc directives. But in particular it is interested in the following block tags:

- `@name name` - the name of the ngdoc document
- `@param {type} name description` - describes a parameter of a function
- `@returns {type} description` - describes what a function returns
- `@requires` - normally indicates that a JavaScript module is required; in an Angular service it is used to describe what other services this service relies on
- `@property` - describes a property of an object

- `@description` - used to provide a description of a component in markdown
- `@link` - specifies a link to a URL or a type in the API reference. Links to the API have the following structure:
  - the module namespace, followed by `.` (optional, default `ng`)
  - the `@ngdoc` type (see below), followed by `:` (optional, automatically inferred)
  - the name
  - the method, property, or anchor (optional)
  - the display name

  For example: `{@link ng.type:$rootScope.Scope#$new Scope.$new()}`.
- `@example` - specifies an example. This can be a simple code block, or a runnable example.
- `@deprecated` - specifies that the following code is deprecated and should not be used. In The AngularJS docs, there are two specific patterns which can be used to further describe the deprecation: `sinceVersion="<version>"` and `removeVersion="<version>"`

The `type` in `@param` and `@returns` must be wrapped in `{}` curly braces, e.g. `{Object|Array}`. Parameters can be made optional by *either* appending a `=` to the type, e.g. `{Object=}`, *or* by putting the `[name]` in square brackets. Default values are only possible with the second syntax by appending `=<value>` to the parameter name, e.g. `@param {boolean} [ownPropsOnly=false]`.

Descriptions can contain markdown formatting.

AngularJS-specific jsdoc directives

In addition to the standard jsdoc tags, there are a number that are specific to the Angular code-base:

- `@ngdoc` - specifies the type of thing being documented. See below for more detail.
- `@eventType emit|broadcast` - specifies whether the event is emitted or broadcast
- `@usage` - shows how to use a `function` or `directive`. Is usually automatically generated.
- `@knownIssue` - adds info about known quirks, problems, or limitations with the API, and possibly, workarounds. This section is not for bugs.

The following are specific to directives:

- `@animations` - specifies the animations a directive supports
- `@multiElement` - specifies if a directive can span over multiple elements
- `@priority` - specifies a directive's priority
- `@restrict` - is extracted to show the usage of a directive. For example, for [E]lement, [A]ttribute, and [C]lass, use `@restrict ECA`
- `@scope` - specifies that a directive will create a new scope

The `@ngdoc` Directive

This directive helps to specify the template used to render the item being documented. For instance, a directive would have different properties to a filter and so would be documented differently. The commonly used types are:

- `overview` - a general page (guide, api index)
- `provider` - AngularJS provider, such as `$compileProvider` or `$httpProvider`.
- `service` - injectable AngularJS service, such as `$compile` or `$http`.
- `object` - well defined object (often exposed as a service)

- `function` - function that will be available to other methods (such as a helper function within the ng module)
- `method` - method on an object/service/controller
- `property` - property on an object/service/controller
- `event` - AngularJS event that will propagate through the `$scope` tree.
- `directive` - AngularJS directive
- `filter` - AngularJS filter
- `error` - minErr error description

General documentation with Markdown

Any text in tags can contain markdown syntax for formatting. Generally, you can use any markdown feature.

Headings

Only use *h2* headings and lower, as the page title is set in *h1*. Also make sure you follow the heading hierarchy. This ensures correct table of contents are created.

Code blocks

In line code can be specified by enclosing the code in back-ticks (`). A block of multi-line code can be enclosed in triple back-ticks (```) but it is formatted better if it is enclosed in <pre>...</pre> tags and the code lines themselves are indented.

Writing runnable (live) examples and e2e tests

It is possible to embed examples in the documentation along with appropriate e2e tests. These examples and scenarios will be converted to runnable code within the documentation. So it is important that they work correctly. To ensure this, all these e2e scenarios are run as part of the continuous integration tests.

If you are adding an example with an e2e test, you should run the test locally first to ensure it passes. You can change `it(...)` to `fit(...)` to run only your test, but make sure you change it back to `it(...)` before committing.

The `<example>` tag

This tag identifies a block of HTML that will define a runnable example. It can take the following attributes:

- `animations` - if set to `true` then this example uses ngAnimate.
- `deps` - Semicolon-separated list of additional angular module files to be loaded, e.g. `angular-animate.js`
- `name` - every example should have a name. It should start with the component, e.g directive name, and not contain whitespace
- `module` - the name of the app module as defined in the example's JavaScript

Within this tag we provide `<file>` tags that specify what files contain the example code.

```
<example
  module="angularAppModule"
  name="exampleName"
  deps="angular-animate.js;angular-route.js"
  animations="true">
  ...
  <file name="index.html">...</file>
```

```
  <file name="script.js">...</file>
  <file name="animations.css">...</file>
  <file name="protractor.js">...</file>
  ...
</example>
```

You can see an example of a well-defined example in the `ngRepeat` documentation.