Developer Tools for Angular

Here you will find a collection of tools and tips for keeping your application perform well and contain fewer bugs.

Angular debug tools in the dev console

Angular provides a set of debug tools that are accessible from any browser's developer console. In Chrome the dev console can be accessed by pressing Ctrl + Shift + j.

Enabling debug tools

By default the debug tools are disabled. You can enable debug tools as follows:

```
import {ApplicationRef} from '@angular/core';
import {platformBrowserDynamic} from '@angular/platform-browser-dynamic';
import {enableDebugTools} from '@angular/platform-browser';


platformBrowserDynamic().bootstrapModule(AppModule)
  .then(moduleRef => {
    const applicationRef = moduleRef.injector.get(ApplicationRef);
    const appComponent = applicationRef.components[0];
    enableDebugTools(appComponent);
  })
```

Using debug tools

In the browser open the developer console (Ctrl + Shift + j in Chrome). The top level object is called `ng` and contains more specific tools inside it.

Example:

```
ng.profiler.timeChangeDetection();
```

Performance

Change detection profiler

If your application is janky (it misses frames) or is slow according to other metrics, it is important to find the root cause of the issue. Change detection is a phase in Angular's lifecycle that detects changes in values that are bound to UI, and if it finds a change it performs the corresponding UI update. However, sometimes it is hard to tell if the slowness is due to the act of computing the changes being slow, or due to the act of applying those changes to the UI. For your application to be performant it is important that the process of computing changes is very fast. For best results it should be under 3 milliseconds in order to leave room for the application logic, the UI updates and browser's rendering

pipeline to fit within the 16 millisecond frame (assuming the 60 FPS target frame rate).

Change detection profiler repeatedly performs change detection without invoking any user actions, such as clicking buttons or entering text in input fields. It then computes the average amount of time it took to perform a single cycle of change detection in milliseconds and prints it to the console. This number depends on the current state of the UI. You will likely see different numbers as you go from one screen in your application to another.

Running the profiler Enable debug tools (see above), then in the dev console enter the following:

```
ng.profiler.timeChangeDetection();
```

The results will be printed to the console.

Recording CPU profile Pass `{record: true}` an argument:

```
ng.profiler.timeChangeDetection({record: true});
```

Then open the "Profiles" tab. You will see the recorded profile titled "Change Detection". In Chrome, if you record the profile repeatedly, all the profiles will be nested under "Change Detection".

Interpreting the numbers In a properly-designed application repeated attempts to detect changes without any user actions should result in no changes to be applied on the UI. It is also desirable to have the cost of a user action be proportional to the amount of UI changes required. For example, popping up a menu with 5 items should be vastly faster than rendering a table of 500 rows and 10 columns. Therefore, change detection with no UI updates should be as fast as possible. Ideally the number printed by the profiler should be well below the length of a single animation frame (16ms). A good rule of thumb is to keep it under 3ms.

Investigating slow change detection So you found a screen in your application on which the profiler reports a very high number (i.e. >3ms). This is where a recorded CPU profile can help. Enable recording while profiling:

```
ng.profiler.timeChangeDetection({record: true});
```

Then look for hot spots using Chrome CPU profiler.

Reducing change detection cost There are many reasons for slow change detection. To gain intuition about possible causes it would help to understand how change detection works. Such a discussion is outside the scope of this document (TODO link to docs), but here are some key concepts in brief.

By default Angular uses "dirty checking" mechanism for finding model changes. This mechanism involves evaluating every bound expression that's active on the UI. These usually include text interpolation via `{{expression}}` and property bindings via `[prop]="expression"`. If any of the evaluated expressions are costly to compute they could contribute to slow change detection. A good way to speed things up is to use plain class fields in your expressions and avoid any kinds of computation. Example:

```
@Component({
  template: '<button [enabled]="isEnabled">{{title}}</button>'
})
class FancyButton {
  // GOOD: no computation, just return the value
  isEnabled: boolean;

  // BAD: computes the final value upon request
  _title: String;
  get title(): String { return this._title.trim().toUpperCase(); }
}
```

Most cases like these could be solved by precomputing the value and storing the final value in a field.

Angular also supports a second type of change detection - the "push" model. In this model Angular does not poll your component for changes. Instead, the component "tells" Angular when it changes and only then does Angular perform the update. This model is suitable in situations when your data model uses observable or immutable objects (also a discussion for another time).