Resolving common technical debt to speed up Angular development

Titas Kurtinaitis | Engineering

Titas Kurtinaitis
05.06.20

Speed up Angular development

Angular development and maintenance projects require a lot of effort—especially when inheriting an existing application with elements needing improvement, and you weren't on the original build team. The agile development team inevitably inherits the pre-existing conditions (aka technical debt) and the decisions made by the original build team. As a developer, how can you fix the technical debt and speed up Angular development? If the primary goal is to create new features, you don't always have the luxury to rewrite the whole application from scratch.

Regardless of the ask, starting point, or current state, there's always an opportunity to tackle debt issues and improve development speed. Apps should continuously evolve and adapt to remain viable. In this article, I'll share the steps our team took to speed up the development of an existing application.

Improve the continuous integration configuration

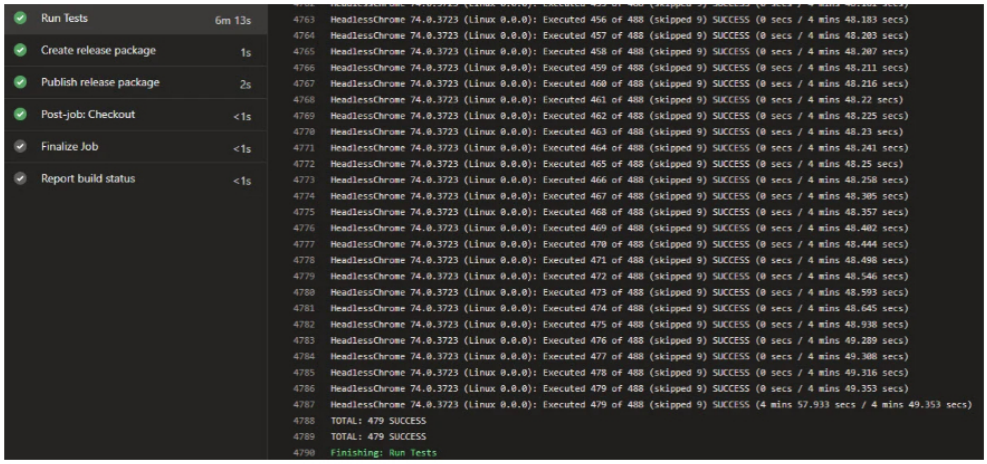
Continuous integration (CI) helps optimize the development process. If this step hasn't been addressed, consider implementing CI as a starting point.

To start, we evaluated the current state of the product, specifically that continuous integration best practices were in place. We found that continuous integration with an automated pull request (PR) builds, all repositories, deployment pipelines were already set up in Azure DevOps. At first glance, the configurations looked decent and well prepared for product development. However, upon further review, we found the system featured many builds (e.g., automated PR builds, commits to feature/* branches trigger individual builds, once PR merged to RC/develop branches separate builds started as well) performing at a slower pace than expected. While we didn't want to reduce the number of builds, we did want to fix the time for the individual builds.

While overall the continuous integration was well prepared, builds were taking around 20 minutes. The unit tests and the npm package install build steps were a bottleneck in our pipelines. The time spent building code before it was ready to be deployed was taking too long. It was clear that we needed to improve the unit tests and npm cache's performance.

How to ensure unit tests run fast

Unit tests are supposed to be cheap and run fast. This was not the case for our team. For example, a unit test step in the build pipeline with around 500 tests ran 5 to 6 minutes on the build machine (~4 minutes tests run itself).



On the local machine, the same suite of tests took 2 to 3 minutes.

```
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 448 of 509 (skipped 9) SUCCESS (0 secs / 1 min 56.993 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 500 of 509 (skipped 9) SUCCESS (2 mins 52.378 secs / 2 mins 49.416 secs)
TOTAL: 500 SUCCESS
TOTAL: 500 SUCCESS
```

Those are huge numbers that got progressively worse with new tests added. After further investigation, we spotted two big testing issues.

ISSUE #1: "TestingCommonModule" created to mock dependencies was mocking all possible dependencies and injecting them into the test. Almost none of the mocks in that module were associated with tests where "TestingCommonModule" was injected.

TO FIX THE PROBLEM: Avoid modules like in this manner and inject/mock only required dependencies for each specific service/component or whatever aspect needing testing.

ISSUE #2: The TestBed configuration was very slow with huge components that had many test suites. For every test run, the configuration recompiled all components.

TO FIX THE PROBLEM: We decided to get rid of our magic "TestingCommonModule" and use the **ng-bullet** library published by **Nikita Yakovenko**. For additional reference, look at an in-depth **analysis** provided by **Nikita Yakovenko**.

Rewriting all the tests required a lot of effort—especially for a project with a high number of tests. We determined that sorting test files by file size and refactoring the biggest ones would take too long. To figure out what tests to rewrite, we opted to log the execution time/test.

To log tests by execution time, we installed a **karma-spec-reporter** library and configured it in a **karma.conf.js** file. Once set up, we identified and addressed the slowest tests.

```
//karma.conf.js
//adjust to your needs
...
  config.set({
    ...
      reporters: ["spec"],
      specReporter: {
        maxLogLines: 5,             // limit number of lines logged per test
        suppressErrorSummary: true, // do not print error summary
        suppressFailed: false,      // do not print information about failed tests
        suppressPassed: false,      // do not print information about passed tests
        suppressSkipped: true,      // do not print information about skipped tests
        showSpecTiming: false,      // print the time elapsed for each spec
        failFast: true              // test would finish with error when a first fail occurs.
      },
      plugins: ["karma-spec-reporter"],
    ...
```

In terms of time and resources, we allocated 1 day and 1 engineer to fix as many unit tests as possible to

improve performance speed. We removed all "TestingCommonModule" usages that didn't require a lot of effort and rewrote the setup for the slowest test suites using a **ng-bullet** library.

The nicest part, configuring everything was easy.

Instead of a configuration like this…

```
beforeEach(async(() => {
    TestBed.configureTestingModule({
        declarations: [ / list of components / ],
        imports: [ / list of imports / ],
        providers: [ / list of providers / ],
    })
    .compileComponents();
}));
```

…we configured this.

```
import { configureTestSuite } from 'ng-bullet';

configureTestSuite(() => {
    TestBed.configureTestingModule({
        declarations: [ / list of components / ],
        imports: [ / list of imports / ],
        providers: [ / list of providers / ],
    })
    .compileComponents();
}));
```

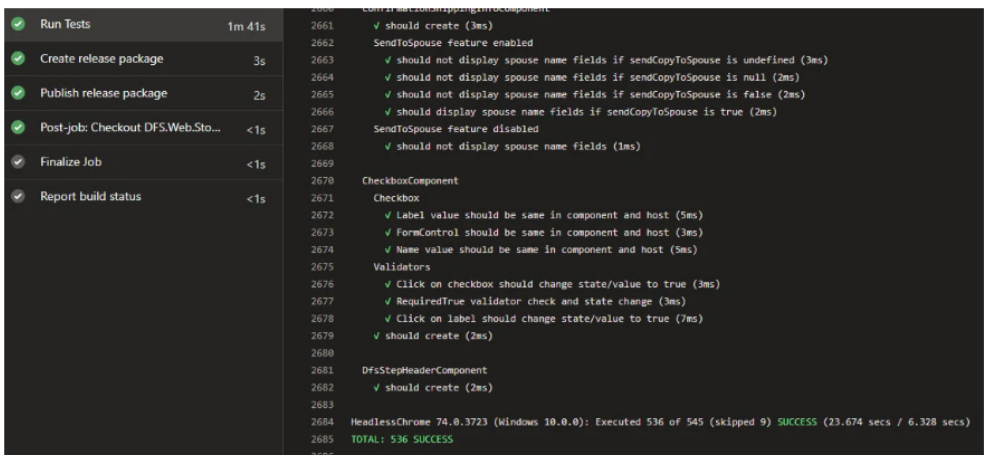
As a result, the test suite cached components instead of recompiling before every test run.

To achieve an even better performance, we ran unit tests in parallel with the **karma-parallel**. We installed the package and configured a **karma.conf.js** file.

```
module.exports = function(config) {
  config.set({
    // NOTE: 'parallel' must be the first framework in the list
    frameworks: ['parallel', 'mocha' /* or 'jasmine' */],
    plugins: [
        // add karma-parallel to the plugins if you encounter something like "karma parallel No provider for fra
        require('karma-parallel'),
        ...
    ],
    parallelOptions: {
      executors: require('os') ? Math.ceil(require('os').cpus().length / 2) : 1,
      shardStrategy: 'round-robin'
    }
  });
};
```

After refactoring most of the tests, the unit test builds step reduced to 1 to 2 minutes (~25 seconds tests' run) without parallel cores.

Unfortunately, the build machine running tests did not have many parallel cores and defaulted to a single core.



```
2687  TOTAL: 536 SUCCESS
2688  Finishing: Run Tests
```

The local machine running tests featured 6 parallel cores.

```
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 98 of 98 SUCCESS (14.679 secs / 2.565 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 69 of 69 SUCCESS (9.931 secs / 2.169 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 92 of 94 (skipped 2) SUCCESS (11.943 secs / 3.005 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 124 of 127 (skipped 3) SUCCESS (11.361 secs / 3.625 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 100 of 104 (skipped 4) SUCCESS (13.498 secs / 3.998 secs)
HeadlessChrome 74.0.3723 (Windows 10.0.0): Executed 50 of 50 SUCCESS (10.709 secs / 2.053 secs)
TOTAL: 533 SUCCESS

TOTAL: 533 SUCCESS
```

Overall, we vastly improved the unit tests run time.

- The unit tests' build step decreased to roughly **300%**.
- The unit test performance locally increased by about **900%**.
- For a team with 20 developers, every developer ran tests once a day. At an hourly rate of $100, it saved nearly **$340**/sprint for 10 working days.

Everyone benefitted. The client saved money. The development team saved time checking code and delivered faster.

How to improve npm cache speed

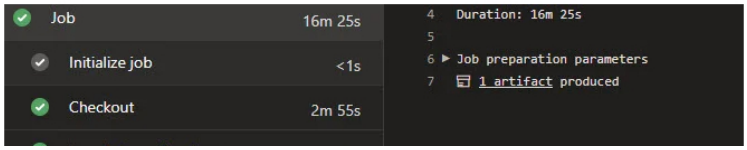
Similar to a unit test, npm caches took too much time. Upon evaluation, we found that the npm package step took about 4 to 6 minutes. This was too long especially given the fact that package.json file did not change often. Reinstalling the same packages over and over again would be a waste of time.

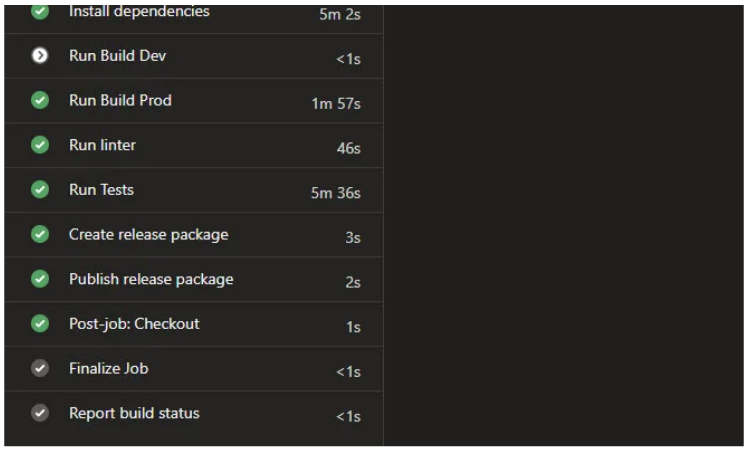
Since the infrastructure used Azure DevOps, we decided to add an **azure-pipelines-artifact-caching-tasks** extension. We first ensured Azure Artifacts was enabled and created a new **Azure Artifacts** feed to store caches before installing the extension. Once fully set up, we added two build steps.

```
- task: 1ESLighthouseEng.PipelineArtifactCaching.RestoreCacheV1.RestoreCache@1
    inputs:
      keyfile: '**/package-lock.json, !**/node_modules/**/package-lock.json, $(Build.SourcesDirectory)/package-l
      targetfolder: '**/node_modules, !**/node_modules/**/node_modules, $(Build.SourcesDirectory)/node_modules'
      vstsFeed: 'e20c93e7-486d-47b0-988b-d5ba25ab5c1e'

  - script: npm ci
    displayName: Install Dependencies
    condition: ne(variables['CacheRestored'], 'true')

  - task: 1ESLighthouseEng.PipelineArtifactCaching.SaveCacheV1.SaveCache@1
    inputs:
      keyfile: '**/package-lock.json, !**/node_modules/**/package-lock.json, $(Build.SourcesDirectory)/package-l
      targetfolder: '**/node_modules, !**/node_modules/**/node_modules, $(Build.SourcesDirectory)/node_modules'
      vstsFeed: 'e20c93e7-486d-47b0-988b-d5ba25ab5c1e'
```

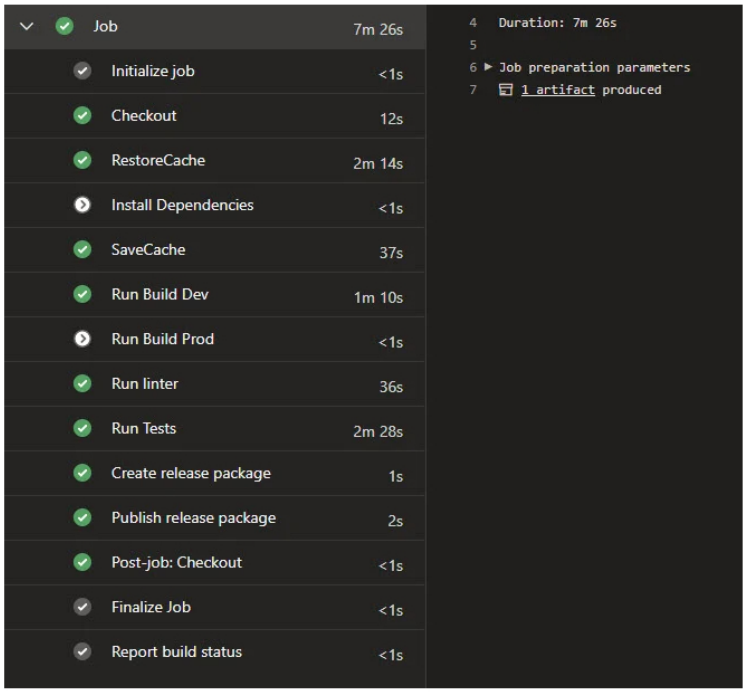
In order to enable an optimistic cache, we included a condition for the "Install Dependencies" step. We set the variable to true once the Restore Cache step was successful. Restoring the previously saved npm cache skips "Install Dependencies" step entirely. These actions significantly improved the build's performance.

BEFORE: ~16 minutes (excluding unit tests improvements)





AFTER: ~8 minutes (including unit tests improvements)



Usually, we averaged ~10 runs per day over an 8-hour period. Caching saved around 80 minutes every day.

NOTE: This particular Azure extension is suited more for UNIX based build agents. You could also set up Windows ones, but since it is executing tar functions, it could require more effort to fully prepare build agents. For the Linux ones, the extension works out of the box.

We reviewed the packages' list—including updating usages, checking the library's functionality use, and removing unused libraries. We found that a few packages used for one simple function could be implemented easily by ourselves. We included the whole library for the simple functionality and used the code in just one place. Removing dead libraries from the code helped reduce the bundle size, speeds up the npm install process, and reduce dependencies in the application.

Streamline wrapping components

Upon taking over the project, we spotted multiple issues with form components. The application was using a third-party library **kendo-UI**, which wasn't the best option. The library usages didn't feature any app layer component to wrap the third-party library and were used incorrectly. The different usages for the same

third-party library component and updating the usages of the components required modifications in multiple places instead of one. Getting rid of the third-party dependency would not be easy.

To resolve these issues, we took the following steps:

**For demo purposes, this example features modifications made to an input component.*

1. We created **form-component-base.ts** for the essential logic needed for the form component.

```
import { EventEmitter, Input, Output } from '@angular/core';
import { ControlValueAccessor, FormControl } from '@angular/forms';

export abstract class FormComponentBase implements ControlValueAccessor {
  @Input() formControl: FormControl;

  @Input() visuallyHiddenLabel: boolean = false;

  @Input() label: string;
  @Input() name: string;

  @Input() labelNgClass: any;
  @Input() inputNgClass: any;

  @Input() required;
  @Input() autofocus;
  @Input() readonly;

  @Output() focus = new EventEmitter();
  @Output() blur = new EventEmitter();

  isDisabled: boolean;

  // this one needed for legacy code usages where ngModelControl is being used
  @Input('ngModelControl') set ngModelControl(control: FormControl) {
    this.formControl = control;
  }
  value: any;
  // endregion

  onFocus(event: FocusEvent) {
    this.focus.emit(event);
  }

  onBlur(event: FocusEvent) {
    this.blur.emit(event);
  }

  propagateChange = (value: any) => {};
  onTouched = () => {};

  writeValue(value: any): void {
    if (this.formControl && value !== undefined) {
      this.value = value;
    }
  }

  registerOnChange(fn: any): void {
    this.propagateChange = fn;
  }

  onChange(): void {
    if (this.formControl && this.formControl.value !== undefined) {
      this.propagateChange(this.formControl.value);
    }
  }

  registerOnTouched(fn: () => void): void {
    this.onTouched = fn;
  }

  setDisabledState?(isDisabled: boolean): void {
    this.isDisabled = isDisabled;
  }
}
```

2. We added **input.component.ts**, which extended the **form-component-base.ts**.

```
import { AfterViewInit, Component, forwardRef, Input } from '@angular/core';
import { NG_VALUE_ACCESSOR } from '@angular/forms';
import { FormComponentBase } from '@@shared/form-controls/form-component-base';

@Component({
  selector: 'sf-input',
  templateUrl: './input.component.html',
  styleUrls: ['./input.component.scss'],
  providers: [
    {
```

```
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => InputComponent),
      multi: true
    }
  ]
})
export class InputComponent extends FormComponentBase implements AfterViewInit {
  @Input() type = 'text';
  @Input() maxLength: number;
  @Input() placeholder: string;
  @Input() useKendo: boolean = false;
  @Input() isDisabled: boolean;
  @Input() useCustomValidationLabel: boolean = false;

  constructor() {
    super();
  }

  ngAfterViewInit(): void {}
}
```

3. We included **input.component.html** to wrap the **kendoTextBox** usage, which rendered kendo-UI input or a custom component depending on the need.

```
<label
  [ngClass]="visuallyHiddenLabel ? 'visually-hidden' : (labelNgClass || 'control-label')"
  [for]="name">
  {{label}}
</label>

<ng-container *ngIf="!useKendo else kendoTmpl">
  <input
    [formControl]="formControl"
    [id]="name"
    [name]="name"
    [type]="type"
    [ngClass]="inputNgClass || 'k-textbox'"
    [attr.placeholder]="placeholder"
    [attr.maxlength]="maxLength"
    [attr.aria-describedby]="name + '-description'"
    [readonly]="readonly"
    [attr.disabled]="isDisabled ? true : null"
    [required]="required"
    [autofocus]="autofocus"
    (input)="onChange()"
    (focus)="onFocus($event)"
    (blur)="onBlur($event)">
</ng-container>

<ng-template #kendoTmpl>
<input
  kendoTextBox
  [formControl]="formControl"
  [id]="name"
  [name]="name"
  [type]="type"
  [ngClass]="inputNgClass || 'k-textbox'"
  [attr.placeholder]="placeholder"
  [attr.maxlength]="maxLength"
  [attr.aria-describedby]="name + '-description'"
  [readonly]="readonly"
  [attr.disabled]="isDisabled ? true : null"
  [required]="required"
  [autofocus]="autofocus"
  (input)="onChange()"
  (focus)="onFocus($event)"
  (blur)="onBlur($event)">
</ng-template>

<ng-content></ng-content>

<sf-validation-message *ngIf="formControl.invalid && formControl.touched" [id]="name + '-description'" [useCusto
```

4. The usage resulted in an **input.component.ts**.

```
<!-- for input with kendo-ui usage -->
<sf-input [useKendo]="true"
          [name]="'scopeValueEnter'"
          [label]="'For'"
          [formControl]="form.get('scopeValueEnter')">
</sf-input>

<!-- for input without kendo-ui usage -->
<sf-input [useKendo]="false"
          [name]="'scopeValueEnter'"
          [label]="'For'"
          [formControl]="form.get('scopeValueEnter')">
</sf-input>
```

These four steps solved the wrapping component issues, keeping the team in sync, and ensuring consistent inputs across the whole application.

Check code quality fast with git hooks

While git hooks seem like an essential element in the development world, not many teams use them. In fact, when we inherited the product, it didn't feature a single git hook. We opted to introduce them to help maintain code quality by running tasks before executing git commands.

Our team established pre-push hooks to allow us to commit elements without any restrictions. Then, once opting to push, a pre-push hook kicked in and ran a set of commands. In our case, those commands validated code formatting and linting rules. For more details on git hook documentation, go **here**.

We used the following tools to set up git hooks quickly and easily.

- **husky**: ran formatting, linting checks, and pre-push hook
- **prettier**: checked code against formatting rules, fixed most of the issues automatically with a few npm commands
- **stylelint** for CSS and **codelyzer;** TSLint for TypeScript: checked linting rules.

NOTE: Tslint was in place, yet needed to use **eslint** because TSlint was already deprecated.

The image below shows how we set up the **package.json**.

```
// husky setup
...
"husky": {
    "hooks": {
      "pre-push": "npm run check-rules"
    }
  },
...

// check-rules command
...
"scripts": {
  ...
  "check-rules": "npm run format-check && npm run lint:scss && npm run lint:ts",
  ...
}
...

// format-check command
...
"scripts": {
  ...
   "format-check": "prettier --config --check ./.prettierrc \"src/{app,environments,assets,scss}/**/*{.ts,.js,.
  ...
}
...

// lint:scss command
...
"scripts": {
  ...
  "lint:scss": "stylelint ./src/**/*.scss --formatter verbose",
  ...
}
...

// lint:ts command
...
"scripts": {
  ...
  "lint:ts": "ng lint",
  ...
}
...
```

The commands and configurations prevented developers from pushing code that didn't meet our

guidelines and rules. This setup helped save a lot of time during code reviews and spot code issues quickly. We didn't waste time arguing about how the code should look. The code style was always being checked, which happened when pushing code. Instead, we focused on the implementation details. If the code style had any issues, they were fixed quickly with the help of the prettier library.

```
// format-fix command
...
"scripts": {
  ...
  "format-fix": "prettier --config --write ./.prettierrc \"src/{app,environments,assets,scss}/**/*{.ts,.js,.jso
  ...
}
...
```

Initially, unit tests ran during the pre-push hook. However, we decided to disable unit tests and allow our build machines to handle them. Forcing unit tests to run every push took time. Our build machines ran them already.

Address debt and increase speed with agile

Too often, inheriting existing products is painful. If code quality is poor, it could demotivate the development team. However, it doesn't have to be. Pinpointing and addressing technical debt inevitably ease the lives of those maintaining or working on the product. Taking the actions covered in this article made maintaining and improving the application easier for our team. We were able to deliver faster and less error-prone code to our clients.

Say goodbye to AngularJS

Start your project