Having fun with Angular and Typescript Transformers

Do you know the burden of handling your RxJs subscriptions manually? Did you ever forget one? Wouldn't it be nice if we never have to think about subscriptions again? Let's explore the options in this article.

Christian Janker 6 June 2019 9 min read

Angular Typescript JavaScript

Do you know the burden of handling your RxJs subscriptions manually? Did you ever forget one? Did you ever run into the problem that you thought you would be safe with the use of the async pipe in the template and then after some time a new requirement comes in and you realise that you need a

subscribe call in your component class as well? This may be a smell for a bad design of some of your components, but let's be honest: Sometimes it is the right way in order to get things done.

> Wouldn't it be nice if we never have to think about subscriptions again?

No more manual handling of subscription arrays.
No more `takeUntil(this.destroyed$)`
No more `subscription.add()`
No more pain and fear ;)

We can achieve that with the help of **Typescript Transformers** at build time.

> Before we go into ecstasy about this, which I certainly did, I have to point out that there are disadvantages with black magic code generation like the one I am presenting here. Sometimes it is even wrong to unsubscribe. So be aware, that the following example primarily has a learning purpose.

What are typescript transformer?

Typescript Transformers allow us to hook into the compilation process of Typescript and transform the Abstract Syntax Tree (AST) that is produced.

This allows us to change the existing code at compile time. In the following sections of this post we will, for example, use it to:

- find all classes with a `@Component()` decorator
- find all calls to `subscribe()` of RxJs
- generate methods like `ngOnDestroy`
- extend the body of existing methods
- ..

This is a very powerful process and it is heavily used by the Angular compilation process itself.

To get a feeling for an AST it helps to take a look at an example at astexplorer.net. On the left side of this explorer you can see the source of the component class `TestComponent` and on the right side its AST representation. You can change the code on the left and the AST is immediately updates. This tool will become incredibly helpful later when we want to find out how to write the transformer code to extend the existing source or generate new parts.

But first, let’s have a look at a basic skeleton of a transformer:

```
function simpleTransformerFactory(context: ts.TransformationContext) {
  // Visit each node and call the function 'visit' from below
  return (rootNode: ts.SourceFile) => ts.visitNode(rootNode, visit);

  function visit(node: ts.Node): ts.Node {
    if (ts.isClassDeclaration(node)) {
      console.log('Found class node! ', node.name.escapedText);
    }

    // Visit each Child-Node recursively with the same visit function
    return ts.visitEachChild(node, visit, context);
  }
}
```

```
// Typings: typescript.d.ts

/**
 * A function that is used to initialize and return a `Transformer` callback, which in turn
 * will be used to transform one or more nodes.
 */
type TransformerFactory<T extends Node> = (context: TransformationContext) => Transformer<T>;

/**
 * A function that transforms a node.
 */
type Transformer<T extends Node> = (node: T) => T;

/**
 * A function that accepts and possibly transforms a node.
 */
type Visitor = (node: Node) => VisitResult<Node>;
type VisitResult<T extends Node> = T | T[] | undefined;
```

In our example the function `simpleTransformerFactory` is the `TransformerFactory`, which returns a `Transformer`.

The typings of Typescript itself (the second code snippet) show that a transformer itself is just a function that takes a `Node` and returns a `Node`.

In the snippet above, where we just log every class name we find, we walk through the Typescript AST (Abstract Syntax Tree) with the so-called visitor pattern **where every node** of the **AST** is being visited.

A node could be a:

- **CallExpression** like `this.click$.subscribe()`
- **BinaryExpression** like `this.subscription = this.click$.subscribe()`
- **ClassDelcaration** like `class Foo {}`
- **ImportDeclaration** like `import {Component} from '@angular/core'`
- **VariableStatement** like `const a = 1 + 2`
- **MethodDeclaration**
- ...

Generating the unsubscribe code

Our goal is that before all the transformers of Angular itself are running our custom transformer is executed. Its task is to find all `subscribe` calls in components and generate the code to automatically unsubscribe from them in the `ngOnDestroy` method.

Thanks to Manfred Steyer and David Kingma there is not that much work to do to achieve that.

In order to be able to inject our custom transformer into the transformation process of an Angular-CLI project, we can use the ngx-build-plus library and its plugin feature. In a plugin, we can access the `AngularCompilerPlugin` and add our transformer to the "private" transformers array.

```
import { unsubscribeTransformerFactory } from './transformer/unsubscribe.transformer';
import { AngularCompilerPlugin } from '@ngtools/webpack';

function findAngularCompilerPlugin(webpackCfg): AngularCompilerPlugin | null {
  return webpackCfg.plugins.find(plugin =>  plugin instanceof AngularCompilerPlugin);
}

// The AngularCompilerPlugin has nog public API to add transformations, user private API
_transformers instead.
function addTransformerToAngularCompilerPlugin(acp, transformer): void {
  acp._transformers = [transformer, ...acp._transformers];
}

export default {
  pre() {},

  // This hook is used to manipulate the webpack configuration
  config(cfg) {
    // Find the AngularCompilerPlugin in the webpack configuration
    const angularCompilerPlugin = findAngularCompilerPlugin(cfg);

    if (!angularCompilerPlugin) {
      console.error('Could not inject the typescript transformer: Webpack AngularCompilerPlugin
not found');
      return;
    }

    addTransformerToAngularCompilerPlugin(angularCompilerPlugin,
unsubscribeTransformerFactory(angularCompilerPlugin));
    return cfg;
  },

  post() {
  }
};
```

The next code block shows the main part of the Typescript Transformer, that is responsible for generating the unsubscribe calls. It is not the whole source of the transformer but it shows the most important steps.

```
export function unsubscribeTransformerFactory(acp: AngularCompilerPlugin) {
  return (context: ts.TransformationContext) => {

    const checker = acp.typeChecker;

    return (rootNode: ts.SourceFile) => {

      let withinComponent = false;
      let containsSubscribe = false;

      function visit(node: ts.Node): ts.Node {

        // 1.
        if (ts.isClassDeclaration(node) && isComponent(node)) {
          withinComponent = true;

          // 2. Visit the child nodes of the class to find all subscriptions first
          const newNode = ts.visitEachChild(node, visit, context);

          if (containsSubscribe) {
            // 4. Create the subscriptions array
            newNode.members = ts.createNodeArray([...newNode.members,
createSubscriptionsArray()]);

            // 5. Create the ngOnDestroyMethod if not there
            if (!hasNgOnDestroyMethod(node)) {
              newNode.members = ts.createNodeArray([...newNode.members,
createNgOnDestroyMethod()]);
            }

            // 6. Create the unsubscribe loop in the body of the ngOnDestroyMethod
            const ngOnDestroyMethod = getNgOnDestroyMethod(newNode);
            ngOnDestroyMethod.body.statements =
ts.createNodeArray([...ngOnDestroyMethod.body.statements, createUnsubscribeStatement()]);
          }

          withinComponent = false;
          containsSubscribe = false;

          return newNode;
        }

        // 3.
        if (isSubscribeExpression(node, checker) && withinComponent) {
          containsSubscribe = true;
          return wrapSubscribe(node, visit, context);
        }

        return ts.visitEachChild(node, visit, context);
      }

      return ts.visitNode(rootNode, visit);
    };
  };
}
```

Step 1
Make sure we are within a component class. If we are, we have to remember that in a context variable `withinComponent` because we just want to enhance `subscribe()` calls that are made within a component.

Step 2
We then immediately call `ts.visitEachChildNode()` to find all subscriptions made in this component.

Step 3
When we find a `subscribe()` expression within a component we wrap it with a `this.subscriptions.push(subsribe-expression)` call.

Step 4
If there was a subscribe expression within the child nodes of the component, we can add the subscriptions array.

Step 5
Then we try to find the `ngOnDestroy` method and create it if there isn't one.

Step6
At last, we extend the body of the `ngOnDestroy` method with the unsubscribe calls:
`this.subscriptions.forEach(s => s.unsubscribe())`

Full Source

Following is the full source of the unsubscribe transformer. I don't want to go into the detail of the Typescript Compiler API itself, because it would be definitely too much for the scope of this post.

My approach basically was a trial and error one. Pasting some existing source code into astexplorer.net and then trying to create the AST programmatically.

I will share some useful links to other transformer posts in the summary section.

```
import * as ts from 'typescript';
import {AngularCompilerPlugin} from '@ngtools/webpack';

// Build with:
// Terminal 1: tsc --skipLibCheck --module umd -w
// Terminal 2: ng build --aot --plugin ~dist/out-tsc/plugins.js
// Terminal 3: ng build --plugin ~dist/out-tsc/plugins.js

const rxjsTypes = [
  'Observable',
  'BehaviorSubject',
  'Subject',
  'ReplaySubject',
  'AsyncSubject'
];

/**
 *
 * ExpressionStatement
 *  -- CallExpression
 *     -- PropertyAccessExpression
 *
 *
 * looking into:
 *    - call expressions within a
 *    - expression statement only
 *    - that wraps another call expression where a property is called with subscribe
 *    - and the type is contained in rxjsTypes
 *
 */
function isSubscribeExpression(node: ts.Node, checker: ts.TypeChecker): node is ts.CallExpression
{
  // ts.isBinaryExpression
  // ts.isCallExpression
  // ts.isClassDeclaration
  // ts.is

  return ts.isCallExpression(node) &&
    node.parent && ts.isExpressionStatement(node.parent) &&
    ts.isPropertyAccessExpression(node.expression) &&
    node.expression.name.text === 'subscribe' &&
    rxjsTypes.includes(getTypeAsString(node, checker));
}

function getTypeAsString(node: ts.CallExpression, checker: ts.TypeChecker) {
  const type: ts.Type = checker.getTypeAtLocation((node.expression as ts.PropertyAccessExpression
| ts.CallExpression).expression);
  console.log('TYPE: ', type.symbol.name);
  return type.symbol.name;
}

/**
 * Takes a subscibe call expression and wraps it with:
 * this.subscriptions.push(node)
 */
function wrapSubscribe(node: ts.CallExpression, visit, context) {
  return ts.createCall(
    ts.createPropertyAccess(
      ts.createPropertyAccess(ts.createThis(), 'subscriptions'),
```

```
      'push'
    ),
    undefined,
    [ts.visitEachChild(node, visit, context)]
  );
}

function logComponentFound(node: ts.ClassDeclaration) {
  console.log('Found component: ', node.name.escapedText);
}

function isComponent(node: ts.ClassDeclaration) {
  return node.decorators && node.decorators.filter(d =>
d.getFullText().trim().startsWith('@Component')).length > 0;
}

/**
 * creates an empty array property:
 * subscriptions = [];
 */
function createSubscriptionsArray() {
  return ts.createProperty(
    undefined,
    undefined,
    'subscriptions',
    undefined,
    undefined,
    ts.createArrayLiteral()
  );
}

function isNgOnDestroyMethod(node: ts.ClassElement): node is ts.MethodDeclaration {
  return ts.isMethodDeclaration(node) && (node.name as ts.Identifier).text == 'ngOnDestroy';
}

function hasNgOnDestroyMethod(node: ts.ClassDeclaration) {
  return node.members.filter(node => isNgOnDestroyMethod(node)).length > 0;
}

function getNgOnDestroyMethod(node: ts.ClassDeclaration) {
  const n = node.members
    .filter(node => isNgOnDestroyMethod(node))
    .map(node => node as ts.MethodDeclaration);
   return n[0];
}

function createNgOnDestroyMethod() {
  return ts.createMethod(
    undefined,
    undefined,
    undefined,
    'ngOnDestroy',
    undefined,
    [],
    [],
    undefined,
    ts.createBlock([], true)
  );
}
```

```
function createUnsubscribeStatement() {
  return ts.createExpressionStatement(
    ts.createCall(
      ts.createPropertyAccess(
        ts.createPropertyAccess(ts.createThis(), 'subscriptions'),
        'forEach'
      ),
      undefined,
      [
        ts.createArrowFunction(
          undefined,
          undefined,
          [
            ts.createParameter(undefined, undefined, undefined, 'sub', undefined, undefined,
undefined)
          ],
          undefined,
          ts.createToken(ts.SyntaxKind.EqualsGreaterThanToken),
          ts.createCall(
            ts.createPropertyAccess(ts.createIdentifier('sub'), 'unsubscribe'),
            undefined,
            []
          )
        )
      ]
    )
  );
}

export function unsubscribeTransformerFactory(acp: AngularCompilerPlugin) {
  return (context: ts.TransformationContext) => {

    const checker = acp.typeChecker;

    return (rootNode: ts.SourceFile) => {

      let withinComponent = false;
      let containsSubscribe = false;

      function visit(node: ts.Node): ts.Node {

        // 1.
        if (ts.isClassDeclaration(node) && isComponent(node)) {
          withinComponent = true;

          // 2. Visit the child nodes of the class to find all subscriptions first
          const newNode = ts.visitEachChild(node, visit, context);

          if (containsSubscribe) {
            // 4. Create the subscriptions array
            newNode.members = ts.createNodeArray([...newNode.members,
createSubscriptionsArray()]);

            // 5. Create the ngOnDestroyMethod if not there
            if (!hasNgOnDestroyMethod(node)) {
              newNode.members = ts.createNodeArray([...newNode.members,
createNgOnDestroyMethod()]);
            }

            // 6. Create the unsubscribe loop in the body of the ngOnDestroyMethod
```

```
            const ngOnDestroyMethod = getNgOnDestroyMethod(newNode);
            ngOnDestroyMethod.body.statements =
ts.createNodeArray([...ngOnDestroyMethod.body.statements, createUnsubscribeStatement()]);
          }

          withinComponent = false;
          containsSubscribe = false;

          return newNode;
        }

        // 3.
        if (isSubscribeExpression(node, checker) && withinComponent) {
          containsSubscribe = true;
          return wrapSubscribe(node, visit, context);
        }

        return ts.visitEachChild(node, visit, context);
      }

      return ts.visitNode(rootNode, visit);
    };
  };
}
```

To run the whole thing we first have to execute following command in our project root:

- `tsc --skipLibCheck --module umd` to compile the `transformer.ts` and the `plugins.ts` file
- then we can run `ng build --plugin ~dist/out-tsc/plugins.js` to execute the build pipeline from Angular with our added plugin. The result of this process can be viewed in the `main.js` file in the dist folder.
- optionally you can serve it with `ng serve --plugin ~dist/out-tsc/plugins.js`

With a given component, in which we intentionally don’t handle our subscriptions:

```
@Component({
  selector: 'app-test',
  templateUrl: './test.component.html',
  styleUrls: ['./test.component.scss']
})
export class TestComponent implements OnDestroy {
  title = 'Hello World';
  showHistory = true;

  be2 = new BehaviorSubject(1);

  constructor(private heroService: HeroService) {
    this.heroService.mySubject.subscribe(v => console.log(v));
    interval(1000).subscribe(val => console.log(val));
  }

  toggle() {
    this.showHistory = !this.showHistory;
  }

  ngOnInit() {
    this.be2.pipe(
      map(v => v)
    ).subscribe(v => console.log(v));
  }

  ngOnDestroy() {
    console.log('fooo');
  }
}
```

Following code is generated after the whole transformation and build process of Angular:

```
var TestComponent = /** @class */ (function () {
    function TestComponent(heroService) {
        this.heroService = heroService;
        this.title = 'Version22: ' + VERSION;
        this.be2 = new rxjs__WEBPACK_IMPORTED_MODULE_1__["BehaviorSubject"](1);
        this.subscriptions = [];
        this.subscriptions.push(this.heroService.mySubject.subscribe(function (v) { return
console.log(v); }));
        this.subscriptions.push(Object(rxjs__WEBPACK_IMPORTED_MODULE_1__["interval"])
(1000).subscribe(function (val) { return console.log(val); }));
    }
    TestComponent.prototype.ngOnInit = function () {
        this.subscriptions.push(this.be2.pipe(Object(rxjs_operators__WEBPACK_IMPORTED_MODULE_3__["map
(function (v) { return v; })).subscribe(function (v) { return console.log(v); }));
    };
    TestComponent.prototype.ngOnDestroy = function () {
        console.log('fooo');
        this.subscriptions.forEach(function (sub) { return sub.unsubscribe(); });
    };
    TestComponent = __decorate([
        Object(_angular_core__WEBPACK_IMPORTED_MODULE_0__["Component"])({
            selector: 'app-test',
            template: __webpack_require__(/*! ./test.component.html */ "./src/app/test.component.html
            styles: [__webpack_require__(/*! ./test.component.scss */ "./src/app/test.component.scss"
        }),
        __metadata("design:paramtypes", [_hero_service__WEBPACK_IMPORTED_MODULE_2__["HeroService"]])
    ], TestComponent);
    return TestComponent;
}());
```

Can you spot the handled subscriptions? :)

Summary

I think there is a reason, why the Angular team keeps its transformer API private. It is a clear sign that we should not extend it on a regular basis.
The unsubscribe transformer is a nice idea but it also shows that the whole thing gets complex very easily because we would have to consider a bunch of edge cases to come up with a bulletproof solution.

Some ideas pop into my mind though:
We could write a custom JAM Stack transformer, which executes http requests at build time.
Or we could leverage the Typescript Compiler API to generate the `TestBed` statement for our unit tests with all necessary dependencies already included.

Further information:

Converting Typescript decorators into static code by Craig Spence
This is a very interesting post, where the access to the AST is simplified by the great library **tsquery**, which you should definitely check out if you want to write custom typescript linters or transformers.

Do you know how Angular transforms your code? by Alexey Zuev
This post explains in detail how Angular itself uses Typescript Transformers at build time. Very informative and we can learn a lot from the transformers written by the Angular team.

Custom Typescript Transformer with Angular by David Kingma
In my opinion an underrated post, which shows how we can write custom transformers and integrate them into the Angular CLI build.

Using the Compiler API
Documentation of the Typescript Compiler

Have a nice day. The sun is shining in Austria. Stay tuned :)

Github Repo of the example above.

Follow me on Twitter.

DISCUSS WITH COMMUNITY

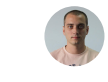
Share

ABOUT THE AUTHOR

Christian Janker

2 stories

Featured articles



Maksym Honchar

19 April 2021 5 min read

How to split HTTP Interceptors between multiple backends

How to use TS decorators to add caching logic to API calls

Maksym Honchar

7 April 2021 6 min read

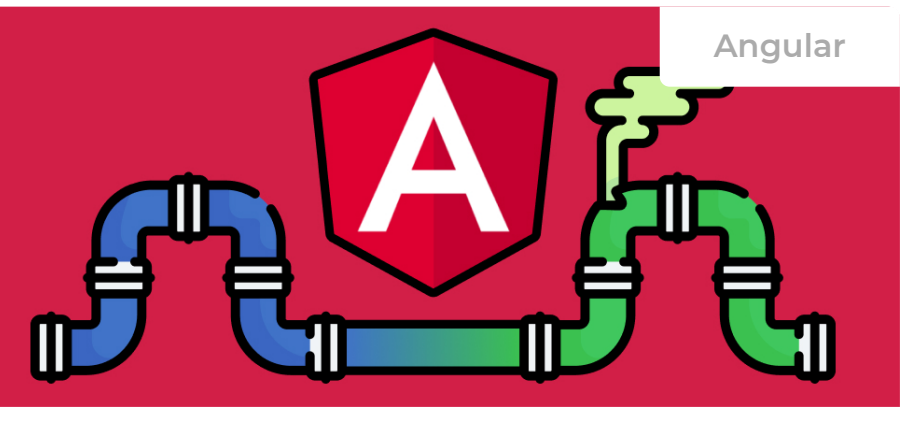
This article explains one of the possible ways to build different types of HttpClients for different feature modules (including non-lazy loaded) and

Read more

NGRX Best Practices

Armen Vardanyan

31 March 2021 7 min read



Dharmen Shah

12 April 2021 6 min read

Different ways to run schematics from another schematics

When we create schematics, we sometimes face a situation where we want to run other existing schematics from the same or external collection. In

Read more

Hien Pham

8 April 2021 4 min read

How pure and impure pipes work in Angular Ivy

Understanding how pipes work under the hood by looking at their implementation details in Ivy

Read more

SIGN UP FOR OUR TOP-NOTCH NEWSLETTER

The Deep Dive

Get the latest coverage of **advanced** web development straight into your inbox. Twice a month.

See previous editions →

FOLLOW US

Name

Enter Your name

E-mail

Enter Your E-mail

What are you interested in?

E.g. JavaScript, Angular, React..

SUBSCRIBE