Advanced TypeScript

Agenda

Datum	Onderwerp
1-2-2017	TypeScript Introduction
15-2-2017	Advanced TypeScript
1-3-2017	Angular Introduction
15-3-2017	Angular Building Blocks
29-3-2017	Components
12-4-2017	Ajax
26-4-2017	Data Entry
10-5-2017	Single Page Applications

Downloads: http://bit.ly/centric-ts2

What are we going to cover?

Bundling with Webpack

Tagged unions

Decorators

Async-Await

Using noImplicitAny, noImplicitThis and strictNullChecks

Unit testing and TypeScript

Bundling with Webpack

Webpack is a very popular module bundler.

- Distributed as a Node.js application using NPM.

The Anugular CLI use Webpack out of the box.

- So does Create-React-App.

Webpack main concepts

Entry

- The **entry** is the file where to start bundling
- There can be multiple entry points

Output

- Where to write the bundled output
- A minimum of **filename** and **path** is required

Loaders

- How individual files are transformed when loaded
- A minimum of **test** and **use** is required for each rule

Plugins

- For performing actions on bundles

Webpack configuration

```
const config = {
  entry: './path/to/my/entry/file.js',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: 'my-first-webpack.bundle.js'
  },
  module: {
    rules: [
      {test: /\.js$/, use: 'babel-loader'}
    ]
  },
  plugins: [
    new webpack.optimize.UglifyJsPlugin(),
    new HtmlWebpackPlugin({template: './src/index.html'})
  ]
};
```

Integrating with other build tools

See the TypeScript documentation:

- https://www.typescriptlang.org/docs/handbook/integrating-with-build-tools.html

Tagged unions

Allows for a number of valid types to be combined.

Each type requires a kind property to differentiate between them.

- Has to be a string literal

Inside a **switch** statement on the differentiator the compiler knows the correct type!

Tagged unions example

```
class Car { model: 'Car'; drive(){} }

class Plane { model: 'Plane'; fly(){} }

type Vehicle = Car | Plane;

function move(vehicle: Vehicle) {
    switch (vehicle.type) {
        case 'Plane':
            // vehicle.drive();
            // error TS2339: Property 'drive' does not exist on type 'Plane'.
            vehicle.fly();
            break;
        case 'Car':
            vehicle.drive();
    }
}
```

Decorators

With **decorators** you can annotate TypeScript classes and their members.

- They are used a lot with Angular development
- It's just a function which is passed the **class**, **property** and a **descriptor** as parameters

Decorator **factories** can be used when a decorator needs to be parameterized.

- Just a function that returns the actual decorator function

Note: Decorators are not yet standardized in ECMAScript and may change.

- They require the **--experimentalDecorators** command line option

Creating a decorator

```
function log(target: any,
  key: string,
  descriptor: PropertyDescriptor) {

  const original = target[key];

  target[key] = function (...args) {
    console.log(`=> ${key}(${args}).`);
    original.call(this, ...args);
  }
  return target;
}
```

Using the decorator

```
class Cat {
  constructor(private name: string) {}

  @log
  eat(food) {
    console.log(
      `${this.name} is eating ${food}.`);
  }
}

const zorro = new Cat('Zorro');
zorro.eat('meat');
```

Async-Await

Using **async** and **await** makes writing asynchronous code much easier.

- Instead of using callbacks and nested functions code can be written like synchronous code

Every function that returns a **promise** can be awaited.

- This must be done in a function marked as async

The feature is based on the C# async/await feature.

Async-Await example

```
async function getMovies() {
  var rsp = await fetch('./movies.json')
  var movies = await rsp.json();
  console.table(movies);
}

getMovies();
```

noImplicitAny

Ensures all variables are declared or resolved to a known type.

- Resolving to **any** is a frequent cause of TypeScript not catching errors

Prevents accidental usage of the **any** type.

- The **any** type can still be used explicitly where needed

noImplicitAny example

```
function add(x: number, y: number) {
    return x + y;
}

// error TS7006:
// Parameter 'y' implicitly has an 'any' type.
function subtract(x: number, y) {
    return x - y;
}
```

noImplicitThis

The type of the **this** variable is not always known and can be inferred as **any**.

This flag causes compiler errors when this is the case.

- Explicitly declare **this** in the function parameters

noImplicitThis example

```
var zorro = {
  name:'Zorro',
  eat(this: {name: string}, food: string) {
    console.log(
      this.name, 'is eating', food);
  }
};

zorro.eat('meat');
```

strictNullChecks

With **strictNullChecks** enabled the compiler checks and complains about potential **null** or **undefined** references.

- Dereferencing null or undefined is one of the most frequent runtime errors

Declaring a type as **any** still allows for **null** and **undefined**.

strictNullChecks

```
class Cat {
    constructor(public name: string) {}
}

function getCat(name) {
    if (name) return new Cat(name);
    return null;
}

function printCat(cat: Cat) {
    // At runtime: Uncaught TypeError:
    // Cannot read property 'name' of undefined
    console.log(cat.name);
}

// error TS2322:
// Type 'Cat | null' is not assignable to type 'Cat'.
var zorro: Cat = getCat('');
printCat(zorro);
```

Unit testing with TypeScript

Type checking catches some errors but not all of them.

- Logic errors still require unit testing.

Unit testing TypeScript with Mocha is easy.

- Many other test runners will work as well

Mocha requires the **ts-node** compiler to be registered for TypeScript.

- And **ts-node** requires the typescript compiler to be installed

Chai works great for assertions.

- Don’t forget to install the **mocha** and **chai** type definitions

Code under test

```
export default function greet(name){
  return `Hello ${name}`;
}
```

The package.json

```
{
  "name": "my-app",
  "version": "1.0.0",
  "main": "main.js",
  "scripts": {
    "test": "mocha --compilers ts:ts-node/register **/*-tests.ts"
  },
  "devDependencies": {
    "@types/chai": "^3.4.34",
    "@types/mocha": "^2.2.39",
    "chai": "^3.5.0",
    "mocha": "^3.2.0",
    "ts-node": "^2.0.0",
    "typescript": "^2.1.5"
  }
}
```

The unit test

```
import 'mocha';
import { expect } from 'chai';
import greet from './greet';

describe('Greet', () => {
  it('should work for Maurice', () => {
    const greeting = greet('Maurice');
    expect(greeting)
      .to.equal('Hello Maurice');
  });
});
```

Conclusion

Webpack is great for bundling the source code.

- Deliver only the code you need to the browser

Use Async-Await where appropriate.

- It makes writing and reading asynchronous code much easier

Use checks like noImplicitAny, noImplicitThis and strictNullChecks.

- They help catch a lot of possible logic errors and missing type declarations

Unit testing of TypeScript code is no harder than regular ECMAScript.