Angular

Internet Services Architectures

Michał Wójcik, Waldemar Korłub

Web frameworks

There is a large number of different solutions for web frameworks executed on client side in web browser:

- Angular,
- React,
- Vue.js,
- Ember.js,
- Meteor,
- Mithril,
- Node.js,
- Polymer,
- Aurelia,
- Backbone.js.

Angular Framework

Angular framework versions:

- Angular 1.x - 2009,
- Angular 2.x - 2016,
  - rewritten from scratch,
  - TypeScript language instead of JavaScript,
  - version 2.x is closer to React than to Angular 1.x:
  - migration 1.x → 2.x requires a lot of code changes;
- Angular 4, 5, 6, 7, 8, ... - 2017.
  - semantic versioning,
  - version 3.x skipped due to discrepancy in the versions of individual components (router package),
  - Google aims to release new version every half year,
  - 18 months of support.

Semantic versioning

major.minor.patch

2.7.3

Version elements:

- patch - bugs fixes compatible with previous version, eg. 2.7.4,
- minor - new functionalities compatible with previous versions, eg. 2.8.0,
- major - changes not backward compatible, eg. 3.0.0.

Framework versions and application

If we want access to **new features and security patches**, we need to **migrate** our application to newer versions of the framework:

- patch, minor - backward compatible changes:
  - application code does not need to be changed in order to work,
- major - changes in code are required in order to work:
  - requires working hours,
  - doesn't mean that changes will be painful for developers,
  - Angular's authors do not plan to rewrite the framework from scratch again;
- each subsequent release introduces new changes:
  - if we give up migration later it will only be more difficult.

TypeScript

TypeScript:

- statically typed language transpiled into JavaScript:
  - the developer works in TypeScript, the browser receives understandable JavaScript;
- offers compatibility with the latest versions of JavaScript (ECMAScript 2020),
- possible transpilation to an older version, eg. ECMAScript 5:
  - at the developer's choice: `tsc --target`,
  - the problem of the standard version and the version of browsers.

TypeScript - basic use

Tool installation:

```
npm install -g typescript
```

File transpilation:

```
tsc file.ts
```

Monitoring file changes:

```
tsc --watch file.ts
```

Modern IDEs offer tool integration.

TypeScript - basic types

```typescript
let done: boolean = false;

let age: number = 42;

let color: number = 0xf00dcc;

let username: string = "ookami";

let list1: number[] = [42, 36, 28];

let list2: Array<number> = [27, 45, 19];
```

TypeScript - enums and tuples

```
let x: [string, number] = ["ookami", 24];

enum Color {Red, Green, Blue};

let c: Color = Color.Green;
```

```
let variable: any = 42;
variable = "ookami";
variable = false;
```

TypeScript - function parameters

```
function greeter(username: string): string {
     return "Hello, " + username;
}
```

```
function greeter(username?: string): string {
    if(username) {
        return "Hello, " + username;
    } else {
        return "Hello!"
    }
}
```

```
function greeter(user: string = "world"): string {
    return "Hello, " + user;
}
```

TypeScript - variable number of arguments

```
function greeter(firstName: string, ...otherNames: string[]): string {
    return "Hello, " + username + " " + otherNames.join(", ");
}

greeter("world", "ookami", "kitsune");
```

TypeScript - classes

```
class Greeter {

    greeting: string;

    constructor(message: string) {
        this.greeting = message;
    }

     greet(): string {
        return "Hello, " + this.greeting;
     }

}

let greeter = new Greeter("world");
let greetings = greeter.greet();
```

TypeScript - classes

```
class Animal {

    public name: string;

    public constructor(name: string) {
        this.name = name;
    }

    public move(distanceInMeters: number) {
        console.log(`${this.name} moved ${distanceInMeters}m.`);
    }

}
```

```
class Snake extends Animal {

    constructor(name: string) {
        super(name);
    }

    move(distanceInMeters = 5) {
        console.log("Slithering...");
        super.move(distanceInMeters);
    }

}
```

TypeScript - visibility levels

Visibility levels:

- public - default, fields and methods visible in other classes,
- protected- visible only in inheritance hierarchy,
- private - visible only in class.

```
class Animal {

    private _name: string;

    get name(): string {
        return this._name;
    }

    set name(name: string) {
        this._name = name;
    }

}
```

TypeScript - abstract classes

```
abstract class Animal {

    abstract makeSound(): void;

    move(): void {
        console.log("move");
    }

}
```

TypeScript - interfaces

```
interface Moveable {

    move(distanceInMeters: number): void;
}
```

```
class Snake implements Moveable {
    move(distanceInMeters = 5) {
        console.log("Slithering...");
    }
}
```

TypeScript - interfaces

```
interface Person {

    firstName: string;

    lastName: string;

    get email(): string;

}
```

```
class Student implements Person {

    firstName: string;

    lastName: string;

    get email(): string {
        return this.firstName
            + "." + this.lastName
            + "@student.pg.edu.pl";
  }

}
```

```
let p1: Person = new Student();
```

Angular - basic usage

Tool installation:

```
npm install -g @angular/cli
```

Creating new project:

```
ng new project-name --routing --style=css
```

Starting application (open in default browser):

```
ng serve --open
```

Project structure

End-to-end tests:

```
e2e/                      (set of end-to-end tests)
    src/                  (end-to-end tests for my-app)
        app.e2e-spec.ts   (tests implementation)
        app.po.ts         (configuration for page elements navigation)
     protractor.conf.js   (test-tool config)
     tsconfig.json        (TypeScript config inherits from workspace)
```

Dependencies:

```
node_modules/    (npm packages)
```

Project structure

Sources:

```
src\                (source files for the root-level application project)
    app\            (component files which application logic and data)
    assets\         (images and other static assets)
    environments\   (build configuration options)
    favicon.ico     (icon for bookmark)
    index.html      (main page)
    main.ts         (application entry point)
    polyfills.ts    (provides polyfill scripts for browser support)
    styles.css      (global styles)
    test.ts         (main entry point for unit tests)
```

Other files:

```
.editorconfig       (config for editors)
.gitignore          (git ignore configuration)
angular.json        (CLI config)
browserslist        (supported browsers)
karma.conf.js       (Karma (test runner) configuration)
package.json        (list of npm packages dependencies)
package-lock.json   (list of resolved npm packages versions)
README.md           (just a readme)
tsconfig.app.json   (project specific TypeScript config)
tsconfig.json       (default TypeScript config)
tsconfig.spec.json  (TypeScript config for tests)
tslint.json         (application-specific TSLint configuration)
```

Modules

Modules:

- Angular applications are divided into modules corresponding to particular functionalities,
- each application has a main module called `AppModule`,
- small applications can have only one module, large applications - hundreds of modules,
- modules defined as classes with the `@NgModule` decorator,
- decorators (functions) allow to attach metadata to a class.

NgModule - metadata

Highlights **metadata** of the `NgModule` decorator:

- `declarations` - a list of components used to build application views,
- `exports` - list of components that should be available for use by other modules,
- `imports` - list of modules whose exported classes are used in the current module,
- `providers` - a list of providers enabling the building of service instances to be used in the entire application (in all modules),
- `bootstrap` - component representing the main view of the application (all other views are loaded into it), used only for `AppModule`.

NgModule - example

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Components

Components:

- represent fragments of views that make up the application interface,
- defined as classes with the `@Component` decorator,
- are represented by additional tags placed in the HTML code, e.g.:

```
<body>
  <app-root></app-root>
</body>
```

Components

Component defines:

- template used to build a view fragment (HTML tags also including tags for other components),
- data for presentation (model),
- behaviors (event handling functions).

`@Component()` - the most important metadata:

- `selector` - a CSS selector that specifies which tags on the page are to be filled with the component's content,
- `templateUrl` - path to the `.html` file with the template,
- `styleUrls` - a list of CSS style files for this components.

`@Input()` - allows to pass attributes to a component from parent.

`@Output()` - allows to pass events to parent from component.

Component - example

```
import {Component, Input, OnInit} from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  @Input()
  name: string;

  value: string;

  constructor() { }

  ngOnInit() {
    this.value = 'Hello ' + this.name + '!';
  }

}
```

```
<app-hello [name]="'world'"></app-hello>
```

Component - example

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {HelloComponent} from './hello/hello.component';

@NgModule({
  declarations: [
    AppComponent,
    HelloComponent // !
  ],
  imports: [
    BrowserModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Templates

Templates:

- used by components to generate content in the browser window,
- the syntax is based on the syntax of the HTML language,
- contain additional elements:
  - tags for attaching other components to the view,
  - directives to control the process of generating the resulting HTML code.

Template - example

```
<div *ngIf="value">
  {{value}}
</div>
```

Data binding

There are 4 **forms of data binding** available:

- value interpolation: `{{...}}`,
- DOM element property binding: `[property]`,
- binding event handler: `(event)`,
- bidirectional data binding: `[(...)]`.

Interpolation

Interpolation:

- allows to determine the value used in the view based on an expression,
- the expression most often refers to fields / properties of a component class,
- expression in parentheses `{{...}}` is converted to a string before being put in the view.

```
<h3>{{imgTitle}}</h3>
<img src="{{imgUrl}}">
```

Expressions

Expressions (template expressions):

- can carry out additional operations,
- should not cause side effects,
- should be quick to make,
- should be as short and simple as possible,
- complex logic should be placed in the component method and called in the expression,
- expression should be idempotent.

```
<p>The threshold has been exceeded {{score - threshold}} points.</p>
```

Property binding

Property binding:

- expression values can also be associated with the properties of DOM tree elements and component properties,
- this bond is unidirectional:
  - changing the value of an expression changes the value of the property, but not vice versa;
- bindings refer to the properties of the DOM tree elements and not to HTML tag attributes.

```
<button [disabled]="unchanged">Cancel</button>
<img [src]="imgUrl">
<app-book-detail [book]="selectedBook">
```

CSS properties binding

CSS properties binding

- the binding object can be CSS classes,
- or individual CSS properties.

```
<div [class.special]="special">...</div>
```

```
<button [style.color]="special ? 'red': 'green'">
<button [style.background-color]="canSave ? 'cyan':'grey'">
```

Event binding

Event binding - enables calling of event handling functions defined in the component class in response to user actions, e.g .:

```
<button (click)="onSave()">Save</button>
```

Event binding

```
import {Component, Input, OnInit} from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  constructor() { }

  ngOnInit() {
  }

  onSave(): void {

  }

}
```

Bidirectional binding

Bidirectional binding:

- changing the value of the associated field changes the value of the property,
- changing the value of a property changes the value of the field,
- especially useful when working with forms,
- built by hand or with `ngModel`.

```
<input [value]="name" (input)="name=$event.target.value" >
```

```
<input [(ngModel)]="name">
```

Component - example

```
import {Component, Input, OnInit} from '@angular/core';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  @Output()
  helloEvent: EventEmitter<string> = new EventEmitter<string>();

  constructor() { }

  ngOnInit() {
  }

  emit(): void {
    this.helloEvent.emit('hello');
  }

}
```

Component - example

```
<app-hello (helloEvent)="onHelloAction($event)"></app-hello>
```

Directives

Directives:

- HTML documents have a static structure,
- Angular view templates are dynamic,
- the resulting HTML is the result of processing the template in accordance with the directives placed in it,
- example directives:
  - `\*ngFor` - adding elements in a loop,
  - `\*ngIf` - displaying the item conditionally.

Services

Services:

- application logic should not be implemented in component classes:
- the component works in the context of a specific view template - it is difficult to reuse the logic embedded in the component class elsewhere in the application,
- the component should define the fields and methods for data binding, and delegate the logic to the services,
- services implement the application logic in a way that is independent of the user interface,
- easy to use in many different contexts,
- services are delivered to components by dependency injection.

Service - example

```
import { Injectable } from '@angular/core';

@Injectable()
export class HelloService {

  constructor() { }

}
```

```
import {Component, Inject, Input, OnInit} from '@angular/core';
import {HelloService} from '../hello.service';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  constructor(private service: HelloService) {
  }

  ngOnInit() {
  }

}
```

Service - example

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {HelloComponent} from './hello/hello.component';
import {HelloService} from './hello.service';

@NgModule({
  declarations: [
    AppComponent,
    HelloComponent
  ],
  imports: [
    BrowserModule
  ],
  providers: [HelloService], // !
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Routing

Routing:

- typical web applications consist of many views between which the user navigates,
- `RouterModule` allows to define addresses that will display selected components (views) of the application,
- The `<base href =" / ">` tag in the `<head>` section of the `index.html` file specifies the base path for addresses within the application.

Routing - example

```
import {NgModule} from '@angular/core';
import {RouterModule, Routes} from '@angular/router';
import {HelloComponent} from './hello/hello.component';

const routes: Routes = [
  {path: 'hello', component: HelloComponent},
  {path: 'hello/:name', component: HelloComponent}
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {
}
```

```
<body>
    <section>
        <router-outlet></router-outlet>
    </section>
</body>
```

The component defined in routing will be displayed below: `<router-outlet></router-outlet>`.

Routing - example

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {HelloComponent} from './hello/hello.component';
import {HelloService} from './hello.service';
import { AppRoutingModule } from './app-routing.module'; // !

@NgModule({
  declarations: [
    AppComponent,
    HelloComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule // !
  ],
  providers: [HelloService],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Navigation

```
<nav class="navbar">
  <ul>
    <li><a routerLink="/hello">Hello</a></li>
  </ul>
</nav>
<section>
  <router-outlet></router-outlet>
</section>
```

Navigation - programmatic

```
import {Component, Input, OnInit} from '@angular/core';
import {ActivatedRoute, Router} from '@angular/router';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  value: string;

  constructor(private route: ActivatedRoute, private router: Router) { // !
  }

  back() {
    this.router.navigateByUrl('/hello'); // !
  }
}
```

Navigation with parameter

```
<nav class="navbar">
  <ul>
    <li><a routerLink="/hello">Hello</a></li>
    <li><a [routerLink]="['/hello', 'ookami']">Hello ookami</a></li>
  </ul>
</nav>
<section>
  <router-outlet></router-outlet>
</section>
```

Navigation with parameter - programmatic

```
import {Component, Input, OnInit} from '@angular/core';
import {ActivatedRoute, Router} from '@angular/router';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  value: string;

  constructor(private route: ActivatedRoute, private router: Router) { // !
  }

  back() {
    this.router.navigateByUrl('/hello');
  }

  refresh() {
    this.router.navigate(['hello', 'ookami']); // !
  }
}
```

Handling navigation parameters

```
import {Component, Input, OnInit} from '@angular/core';
import {ActivatedRoute, Router} from '@angular/router';

@Component({
  selector: 'app-hello',
  templateUrl: './hello.component.html',
  styleUrls: ['./hello.component.css']
})
export class HelloComponent implements OnInit {

  value: string;

  constructor(private route: ActivatedRoute, private router: Router) { // !
  }

  ngOnInit() {
    const id = this.route.snapshot.paramMap.get('name'); // !
    if (id == null) {
      this.value = 'Hello!';
    } else {
      this.value = 'Hello ' + id + '!';
    }
  }

}
```

Calling web services

Use of web services:

- data presented in the front-end application is typically downloaded from the server (from the back-end),
- user input is saved on the server
  - data from the forms on the website, the contents of the basket / order, etc.;
- the back-end application provides its functions in the form of web services:
  - e.g. in REST architecture;
- the `HttpClient` service allows sending HTTP requests to the back-end.

Calling web service - example

```
import {BrowserModule} from '@angular/platform-browser';
import {NgModule} from '@angular/core';

import {AppComponent} from './app.component';
import {HelloComponent} from './hello/hello.component';
import {HelloService} from './hello.service';
import {AppRoutingModule} from './app-routing.module';
import {HttpClientModule} from '@angular/common/http'; // !

@NgModule({
  declarations: [
    AppComponent,
    HelloComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule // !
  ],
  providers: [HelloService],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Calling web service - example

To avoid setting the service address permanently, it is worth using the `proxy.conf.json` file:

```
{
  "/api": {
    "target": "http://localhost:8080/library/",
    "secure": false,
  }
}
```

Calling web service - example

```
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http';
import {Observable} from 'rxjs';
import {Book} from '../model/book';
import {Author} from '../model/author';

@Injectable()
export class BookService {

  constructor(private http: HttpClient) {
  }

  findAllBooks(): Observable<Book[]> {
    return this.http.get<Book[]>('api/books');
  }

}
```

ng commands

Instead of creating individual elements manually, you can use the appropriate commands.

Create a regular class (model):

```
ng generate class model/book
```

Creation of the enum type:

```
ng generate enum model/cover
```

Component creation:

```
ng generate component component/BookList
```

Service creation:

```
ng generate service service/book
```

ng commands

Creation of a routing module:

```
ng generate module app-routing --flat --module=app
```

Starting the server:

```
ng serve
```

Starting the server with the use of a proxy file:

```
ng serve --proxy-config proxy.conf.json
```

Download dependencies in a project without `node_modules`:

```
npm install
```

ng commands

It is a good idea to put the proxy configuration in the `package.json` file, which is used by e.g. IntelliJ:

```
{
  "name": "angular",
  "version": "0.0.0",
  "scripts": {
    "ng": "ng",
    "start": "ng serve --proxy-config proxy.conf.json",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint",
    "e2e": "ng e2e"
...
```