Содержание

Глава 7

Службы, внедрение зависимостей и класс `NgModule`

В этой главе даются следующие рецепты.

- Внедрение простой службы в компонент
- Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`
- Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`
- Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`
- Создание настраиваемой поставщиком службы с помощью свойства `useFactory`

Введение

В версии Angular 1 предоставлялись самые разные виды служб. Многие из них по большей части перекрывали своими возможностями друг друга и были неудобны. И все они были одиночками.

Такой принцип создания служб был полностью отвергнут в версии Angular 2. Вместо этого появилась совершенно новая система внедрения зависимостей, которая оказалась намного более удобной и расширяемой, чем ее предшественница. Она дает в распоряжение разработчиков как атомарные, так и неатомарные типы служб, фабрики, средства совмещения имен и самые разные чрезвычайно полезные инструментальные средства для применения в их приложениях.

Если вам требуется применять службы таким же образом, как и прежде, вы сможете легко перенести свое представление о типах служб и в новую систему. Но тем разработчикам, которым требуется извлечь наибольшую пользу из своих приложений, новая система внедрения зависимостей принесет немалые выгоды в написании масштабируемых приложений.

Внедрение простой службы в компонент

Самым типичным примером может служить внедрение службы в сам компонент. Несмотря на то что порядок определения типов служб и внедрения зависимостей остается в основном прежним, очень важно иметь ясное представление об основах системы внедрения зависимостей в Angular 2, поскольку она имеет ряд заметных отличий.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/4263/**.

Подготовка

Допустим, имеется приведенная ниже заготовка приложения. Цель состоит в том, чтобы реализовать службу, которую можно внедрить в данный компонент и возвратить название статьи, заполнив шаблон.

```
[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;
  constructor() {}
  fillArticle() {}
}
```

Реализация

Как и следовало ожидать, службы в Angular 2 представлены в виде классов. Аналогично компонентам, службы обозначаются декоратором `@Injectable`. Итак, создайте службу в отдельном файле следующим образом:

```
[app/article.service.ts]

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

@Injectable()
export class ArticleService {
  private title_:string = `
    CFO Yodels Quarterly Earnings Call, Stock Skyrockets
  `
}
```

У данной службы имеется закрытое название, которое требуется перенести в компонент, но прежде саму службу необходимо сделать доступной для компонента. Для этого достаточно импортировать службу, а затем указать ее в свойстве `providers` из модуля приложения, как выделено ниже полужирным шрифтом.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleService} from './article.service';

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

А теперь, когда можно предоставить службу, внедрите ее в компонент следующим образом:

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'root',
```

```
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;

  constructor(private articleService_:ArticleService) {}

  fillArticle() {}
}
```

В приведенном выше коде создается новый экземпляр службы типа `ArticleService` при получении экземпляра типа `RootComponent`, а затем данная служба внедряется в конструктор. Все, что внедряется в компонент, будет доступно в виде его члена, которым можно воспользоваться для подключения метода службы к методу компонента, как показано ниже.

```
[app/article.service.ts]

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

@Injectable()
export class ArticleService {
  private title_:string = `
    CFO Yodels Quarterly Earnings Call, Stock Skyrockets
  `;
  getTitle() {
    return this.title_;
  }
}
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="fillArticle()">Show article</button>
    <h2>{{title}}</h2>
  `
})
export class RootComponent {
  title:string;

  constructor(private articleService_:ArticleService) {}
```

```
  fillArticle():void {
    this.title = this.articleService_.getTitle();
  }
}
```

Принцип действия

В отсутствие декоратора созданная только что служба довольно проста по своей композиции. Благодаря декорации `@Injectable()` класс обозначается в Angular как доступный для внедрения в любом другом месте.

В связи с обозначением компонентов как внедряемых возникает целый ряд вопросов, которые следует иметь в виду, чтобы отличать их от компонентов, передаваемых в качестве параметров. В частности, когда получается экземпляр внедряемого класса? Как он связывается с экземпляром компонента? Каким образом контролируются глобальные и локальные экземпляры? Все подобные вопросы обсуждаются более подробно в рецептах, следующих далее в этой главе.

Обозначение службы как внедряемой — это лишь часть общей картины. Компонент еще нужно известить о существовании службы. Сначала следует импортировать класс службы в модуль компонента, но одного этого явно недостаточно. Напомним, что синтаксис, применяемый для внедрения службы, просто служит для того, чтобы указать ее в качестве параметра конструктора. А внутренний механизм Angular достаточно развит логически, чтобы распознавать эти параметры и аргументы как внедряемые компоненты. Но для этого требуется завершающая общую картину часть, чтобы соединить импортированный модуль с тем местом, где он будет служить в качестве внедряемого ресурса.

Эта завершающая часть принимает форму свойства `providers` в определении класса `NgModule`. Для целей данного рецепта совсем не важно знать особенности этого свойства. Но если кратко, то оно представляет массив, где параметр конструктора `articleService` обозначается как внедряемый, а служба типа `ArticleService` должна быть внедрена в конструктор.

Дополнение

Следует признать, что в данном рецепте декораторы языка TypeScript помогают устанавливать внедрение зависимостей. Декораторы модифицируют не экземпляр, а скорее определение класса. Экземпляр типа `NgModule`, где содержится список поставщиков, будет инициализирован прежде любого другого получаемого экземпляра конкретного компонента. Таким образом, каркасу Angular станут известны все службы, которые, возможно, придется внедрить в конструктор.

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.

Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`

Версия Angular 2 заметно отличается от версии Angular 1.x наличием иерархической системы внедрения зависимостей. Из этого обстоятельства вытекает немало существенных последствий, и к самым примечательным из них относится способность контролировать, когда и сколько служб создается.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/2102/`**.

Подготовка

Допустим, имеется следующая заготовка приложения:

```
[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <article></article>
    <article></article>
  `
})
export class RootComponent {}

[app/article.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `
})
export class ArticleComponent {}

[app/article.service.ts]
```

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

@Injectable()
export class ArticleService {
  constructor() {
    console.log('ArticleService constructor!');
  }
}
```

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component;

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

Цель состоит в том, чтобы внедрить единственный экземпляр службы типа `ArticleService` в два порожденных компонента. В данном рецепте метод `console.log()`, вызываемый в конструкторе класса `ArticleService`, позволяет выяснить, когда именно получен отдельный экземпляр службы данного типа.

Реализация

Начните с импорта службы в класс `AppModule`, а затем предоставьте ее следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component;
import {ArticleService} from './article.service;

@NgModule({
```

```
  imports: [
    BrowserModule,
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  providers: [
    ArticleService
  ]
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

Служба типа `ArticleService` предоставляется в том же самом модуле, где объявлен компонент `ArticleComponent`, поэтому службу типа `ArticleService` можно внедрить теперь в порожденные экземпляры типа `ArticleComponent`, как показано ниже. Таким образом, один и тот же экземпляр службы внедряется в порожденные компоненты, поскольку метод `console.log()` выполняется в конструкторе класса `ArticleService` лишь один раз.

```
[app/article/article.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `
})
export class ArticleComponent {
  constructor(private articleService_:ArticleService) {}
}
```

Разделение корневого модуля

По мере расширения приложения остается все меньше оснований втискивать все подряд в один и тот же модуль верхнего уровня. Вместо этого было бы идеально разбить модули на отдельные значимые части. А в данном рецепте было бы предпочтительно предоставить службу типа `ArticleService` тем частям приложения, где их требуется внедрить.

Определите далее новый исходный файл для модуля `ArticleModule` и перенесите в него операторы импорта соответствующих модулей, как показано ниже.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  providers: [
    ArticleService
  ],
  bootstrap: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Затем импортируйте весь этот модуль в модуль `AppModule` следующим образом:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleModule} from './article.module';

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

Остановившись на этом, вы не обнаружите в данном коде ошибок, тем не менее, модуль `AppModule` не в состоянии визуализировать компонент `ArticleComponent`. Дело в том, что в модулях Angular, как и в других

модульных системах, необходимо явно определить, что именно экспортируется в другие модули. Именно это и выделено ниже полужирным шрифтом. Таким образом, экземпляр службы типа `ArticleService` получается лишь один раз.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  providers: [
    ArticleService
  ],
  bootstrap: [
   ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}
```

Принцип действия

Иерархическая структура системы внедрения зависимостей в Angular 2 выгодно используется для предоставления и внедрения служб. Там, где служба внедрена, средствами Angular будет получен экземпляр данной службы, где бы она ни предоставлялась. В определении модуля это происходит лишь один раз.

В данном случае служба типа `ArticleService` предоставляется обоим модулям: `AppModule` и `ArticleModule`. Несмотря на то что данная служба внедряется дважды (по одному разу в каждом экземпляре типа `ArticleComponent`), объявление свойства `providers` в Angular используется с целью решить, когда именно следует создавать службу.

Дополнение

В данный момент у любознательного разработчика может возникнуть немало вопросов, касающихся поведения системы внедрения зависимостей. Имеются многочисленные разновидности конфигурации, которые могут

оказаться полезными для разработчиков, и эти конфигурации требуют лишь минимальной коррекции кода из предыдущего результата.

Внедрение других экземпляров служб в разные компоненты

Как следует из предыдущего пояснения, рассматриваемое здесь приложение можно переконфигурировать, чтобы внедрить другие экземпляры типа `ArticleService` в каждый из порожденных компонентов. Для этого достаточно перенести объявление свойства `providers` из определения модуля в определение класса `ArticleComponent`, как выделено ниже полужирным шрифтом. Оба полученных экземпляра можно проверить, наблюдая результаты, выводимые при вызове метода `console.log()` из конструктора класса `ArticleService`.

```
[app/article.module.ts]

import {NgModule} from '@angular/core';
import {ArticleComponent} from './article.component';

@NgModule({
  declarations: [
    ArticleComponent
  ],
  bootstrap: [
    ArticleComponent
  ],
  exports: [
    ArticleComponent
  ]
})
export class ArticleModule {}

[app/article.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'article',
  template: `
    <p>Article component!</p>
  `,
  providers: [
    ArticleService
  ]
})
export class ArticleComponent {
  constructor(private articleService_:ArticleService) {}
}
```

Получение экземпляра службы

Местоположение поставщиков означает также, что получение экземпляра службы привязано к времени жизни компонента. Для данного приложения это означает, что всякий раз, когда создается компонент, будет получен новый экземпляр службы, если она предоставляется.

Так, если попытаться создать порожденный компонент с предоставляемой в нем службой типа `ArticleService`, то всякий раз, когда произойдет создание компонента `ArticleComponent`, будет создана и новая служба типа `ArticleService`, как показано ниже. Можете сами убедиться, что новые экземпляры данной службы получаются всякий раз, когда вычисляется логическое значение `true` директивы `ngIf`, наблюдая результаты, выводимые при дополнительных вызовах метода `console.log()` из конструктора класса `ArticleService`.

```
[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <h1>root component!</h1>
    <button (click)="toggle=!toggle">Toggle</button>
    <article></article>
    <article *ngIf="toggle"></article>
  `
})
export class RootComponent {}
```

См. также

- Рецепт “Внедрение простой службы в компонент” с пояснениями основ системы внедрения зависимостей в Angular 2.
- Рецепт “Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`”, в котором показано, как перехватывать запросы от поставщиков внедрения зависимостей.

Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`

По мере усложнения разрабатываемого приложения может возникнуть ситуация, когда службами требуется воспользоваться в полиморфном стиле.

Так, в некоторых местах приложения может возникнуть потребность запросить службу А. Но конфигурация в какой-нибудь другой части приложения приведет к тому, что приложение фактически предоставит службу Б. В данном рецепте демонстрируется один полезный способ, но такое поведение позволяет расширить приложение несколькими способами.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/1109/**.

Подготовка

Допустим, что разработка приложения начинается с приведенного ниже каркаса.

Двойные службы

Начните с двух служб, представленных классами `ArticleService` и `EditorArticleService`, а также их интерфейса `ArticleSourceInterface`. В частности, класс `EditorArticleService` наследует от класса `ArticleService`, как показано ниже.

```
[app/article-source.interface.ts]

export interface ArticleSourceInterface {
  getArticle():Article
}

export interface Article {
  title:string,
  body:string,
  // Знак ? обозначает необязательное свойство
  notes?:string
}

[app/article.service.ts]

import {Injectable} from '@angular/core';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class ArticleService implements ArticleSourceInterface {
  private title_:string =
      "Researchers Determine Ham Sandwich Not Turing Complete";

  private body_:string =
          "Computer science community remains skeptical";

  getArticle():Article {
```

```
    return {
      title: this.title_,
      body: this.body_
    };
  }
}
```

[app/editor-article.service.ts]

```
import {Injectable} from '@angular/core';
import {ArticleService} from './article.service';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class EditorArticleService extends ArticleService
       implements ArticleSourceInterface {
  private notes_:string = "Swing and a miss!";

  constructor() {
    super();
  }

  getArticle():Article {
    // объединить объекты и возвратить соединенный объект
    return Object.assign(
      {},
      super.getArticle(),
      {
        notes: this.notes_
      });
  }
}
```

Единообразный компонент

Цель состоит в том, чтобы воспользоваться следующим компонентом и внедрить обе упомянутые выше службы в этом компоненте:

[app/article.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';
import {Article} from './article-source.interface';

@Component({
  selector: 'article',
  template: `
    <h2>{{article.title}}</h2>
    <p>{{article.body}}</p>
    <p *ngIf="article.notes">
```

```
      <i>Notes: {{article.notes}}</i>
    </p>
  `
})
export class ArticleComponent {
  article:Article;
  constructor(private articleService_:ArticleService) {
    this.article = articleService.getArticle();
  }
}
```

Реализация

При указании поставщиков в Angular 2 допускается объявлять ссылку с назначенным псевдонимом, обозначающую, какую именно службу следует предоставить, когда запрашивается один из определенных типов. А поскольку система внедрения зависимостей в Angular 2 последует вверх по иерархическому дереву компонентов, чтобы обнаружить поставщика, то объявить такой псевдоним можно, например, заключив компонент в родительский компонент, где этот псевдоним будет указан, как показано ниже.

```
[app/default-view.component.ts]

import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'default-view',
  template: `
    <h3>Default view</h3>
    <ng-content></ng-content>
  `,
  providers: [ArticleService]
})
export class DefaultViewComponent {}

[app/editor-view.component.ts]

import {Component } from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'editor-view',
  template: `
    <h3>Editor view</h3>
    <ng-content></ng-content>
  `,
  providers: [
```

```
    {provide: ArticleService, useClass: EditorArticleService}
  ]
})
export class EditorViewComponent {}
```

Следует заметить, что оба приведенных выше класса действуют в качестве передаваемых компонентов. Помимо добавления заголовка, что делается лишь в целях пояснения данного рецепта, в этих классах указывается только поставщик безразлично к их содержимому.

Определив классы-оболочки, можно добавить их сначала в модуль приложения, а затем воспользоваться ими для создания двух экземпляров типа `ArticleComponent`, как выделено ниже полужирным шрифтом. Таким образом, заметки получаются в редакторской версии компонента `ArticleComponent`, тогда как в стандартной версии они отсутствуют.

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {DefaultViewComponent} from './default-view.component';
import {EditorViewComponent} from './editor-view.component';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent,
    DefaultViewComponent,
    EditorViewComponent
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}

[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
```

```
  selector: 'root',
  template: `
    <default-view>
      <article></article>
    </default-view>
    <hr />
    <editor-view>
      <article></article>
    </editor-view>
  `
})
export class RootComponent {}
```

Принцип действия

Вид службы, которую предполагалось внедрить, указывался в Angular 1 непосредственно в качестве параметра функции. Например, в результате вызова `function(Article)` внедрялась служба `Article`, а в результате вызова `function(User)` — служба `User`. Это приводило к таким побочным явлениям, как обеспечение минимальной защиты конструкторов, которым нужно было передавать массив символьных строк и соответствующих им функций вида `['Article', function(Article) {}]`.

Теперь все это уже не так. После регистрации поставщика в свойстве `useClass` применяется система сопоставления внедряемых зависимостей, состоящая из двух частей и принятая в Angular 2. К первой части относится поставщик маркеров, т.е. задаваемый в качестве параметра вид внедряемой службы. В данном случае в качестве параметра `private articleService_:ArticleService` задается маркер для запроса внедряемой службы типа `ArticleService`. Этот маркер сопоставляется с поставщиками из иерархии компонентов. Если маркер совпадает, вторая часть данной системы, т.е. сам поставщик, используется для внедрения экземпляра службы.

В действительности объявление `providers: [ArticleService]` является сокращением объявления `providers: [{provide:ArticleService, useClass: ArticleService}]`. Это удобное сокращение, поскольку оно почти всегда обозначает запрос класса сервера, который должен совпадать с внедряемым классом. Но в данном рецепте Angular 2 настраивается на распознавание маркера службы типа `ArticleService`, а следовательно, на применение поставщика службы типа `EditorArticleService`.

Дополнение

Внимательному читателю должно быть теперь ясно, что применение свойства `useClass` ограничено в том смысле, что оно не позволяет независимо контролировать, где именно предоставляется конкретная служба. Иными

словами, место, где перехватывается определение поставщика в свойстве useClass, служит также местом, где предоставляется заменяющий класс.

В данном примере свойство useClass оказывается пригодным потому, что оно дает идеальную возможность предоставить службу типа EditorArticleService в том же указанном месте, где она должна быть заменена службой типа ArticleService. Хотя нетрудно представить ситуацию, когда было бы желательно указать вид заменяющей службы, но внедрить ее выше по иерархии дерева компонентов. Ведь это позволило бы повторно пользоваться экземплярами службы вместо того, чтобы создавать новую службу для каждого объявления свойства useClass.

С этой целью можно воспользоваться свойством useExisting. Оно требует указать вид предоставляемой службы отдельно, но позволяет повторно пользоваться предоставленным ее экземпляром вместо того, чтобы создавать новый. Поэтому можете переконфигурировать только что созданное приложение, чтобы воспользоваться в нем свойством useExisting и предоставить оба рассматриваемых здесь вида служб на уровне компонента RootComponent.

Чтобы проверить правильность вашего представления о поведении служб, удвойте число компонентов Article и добавьте метод log() в конструктор класса ArticleService, чтобы убедиться в создании только одного экземпляра каждой службы, как показано ниже.

```
[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <default-view>
      <article></article>
    </default-view>
    <editor-view>
      <article></article>
    </editor-view>
    <default-view>
      <article></article>
    </default-view>
    <editor-view>
      <article></article>
    </editor-view>
  `
})
export class RootComponent {}

[app/article.service.ts]
```

```
import {Injectable} from '@angular/core';
import {Article, ArticleSourceInterface}
        from './article-source.interface';

@Injectable()
export class ArticleService implements ArticleSourceInterface {
  private title_:string =
     "Researchers Determine Ham Sandwich Not Turing Complete";
  private body_:string =
     "Computer science community remains skeptical";
  constructor() {
    console.log('Instantiated ArticleService!');
  }
  getArticle():Article {
    return {
      title: this.title_,
      body: this.body_
    };
  }
}
```

[app/editor-view.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'editor-view',
  template: `
    <h3>Editor view</h3>
    <ng-content></ng-content>
  `,
  providers: [
    {provide: ArticleService, useExisting: EditorArticleService}
  ]
})
export class EditorViewComponent {}
```

[app/default-view.component.ts]

```
import {Component} from '@angular/core';
import {ArticleService} from './article.service';

@Component({
  selector: 'default-view',
  template: `
    <h3>Default view</h3>
    <ng-content></ng-content>
  `
  // удаленные поставщики
```

```
})
export class DefaultViewComponent {}
```

Если в данной конфигурации применяется свойство `useClass`, то в конечном счете создается один экземпляр службы типа `ArticleService` и два экземпляра службы типа `EditorArticleService`. А после замены на свойство `useExisting` создается лишь один экземпляр каждой из этих служб.

Таким образом, в переконфигурированной версии данного рецепта рассматриваемое здесь приложение выполняет следующие действия.

- На уровне компонента `RootComponent` предоставляется служба типа `EditorArticleService`.
- На уровне компонента `EditorViewComponent` маркеры внедрения службы типа `ArticleService` переадресовываются службе типа `EditorArticleService`.
- На уровне компонента `ArticleComponent` служба типа `ArticleService` внедряется по своему маркеру.

Реорганизация кода с помощью директивных поставщиков

Если рассматриваемая здесь реализация покажется вам неуклюжей и многословной, значит, вы верно понимаете суть дела. Промежуточные компоненты вполне справляются со своими обязанностями, но эти обязанности не выходят за рамки вставки объявления промежуточного поставщика в виде прокладки. Вместо того чтобы заключать объявление поставщика в оболочку компонента, соответствующий оператор можно перенести в директиву и тем самым избавиться от обоих компонентов представлений, как выделено ниже полужирным шрифтом. В итоге приложение будет работать точно так же!

```
[app/root.component.ts]

import {Component} from '@angular/core';
import {ArticleComponent} from './ article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Component({
  selector: 'root',
  template: `
    <article></article>
    <article editor-view></article>
    <article></article>
    <article editor-view></article>
  `
})
export class RootComponent {}
```

```
[app/editor-view.directive.ts]

import {Directive } from '@angular/core';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@Directive({
  selector: '[editor-view]',
  providers: [
    {provide: ArticleService, useExisting: EditorArticleService}
  ]
})
export class EditorViewDirective {}

[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {DefaultViewComponent} from './default-view.component';
import {EditorViewDirective} from './editor-view.directive';
import {ArticleComponent} from './article.component';
import {ArticleService} from './article.service';
import {EditorArticleService} from './editor-article.service';

@NgModule({
  imports: [
  BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent,
    DefaultViewComponent,
    EditorViewDirective
  ],
  providers: [
    ArticleService,
    EditorArticleService
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

См. также

- Рецепт “Внедрение простой службы в компонент” с пояснениями основ системы внедрения зависимостей в Angular 2.

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`", в котором показано, как пользоваться маркерами внедряемых зависимостей для внедрения обобщенных объектов.
- Рецепт "Создание настраиваемой поставщиком службы с помощью свойства `useFactory`" с подробным описанием процесса установки фабрики служб для формирования определений настраиваемых служб.

Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`

В версии Angular 1 предоставлялся на выбор обширный ряд типов служб, которыми можно пользоваться в приложении. К их числу относятся типы служб, позволяющие внедрять статическое значение вместо экземпляра службы. Эта удобная возможность сохранилась и в версии Angular 2.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **`http://ngcookbook.herokuapp.com/3032/`**.

Подготовка

Допустим, имеется следующая заготовка приложения:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  bootstrap: [
```

```
    RootComponent
  ]
})
export class AppModule {}

[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <article></article>
  `
})
export class RootComponent {}

[app/article.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'article',
  template: `
    <img src="{{logoUrl}}">
    <h2>Fool and His Money Reunited at Last</h2>
    <p>Author: Jake Hsu</p>
  `
})
export class ArticleComponent {}
```

Реализация

Несмотря на то что для внедрения отдельного значения формальное объявление класса службы и обозначение декоратора `@Injectable` больше не требуется, сопоставление маркера с поставщиком по-прежнему необходимо. А поскольку класс, предназначенный для приведения типа внедряемого значения больше недоступен, то вместо него должно действовать нечто иное.

Это затруднение разрешается в Angular 2 с помощью класса `OpaqueToken`. Модуль, который представляет этот класс, позволяет создавать бесклассовый маркер, которым можно воспользоваться для соединения внедряемого значения в паре с аргументом конструктора. Наряду с этим можно воспользоваться свойством `useValue`, непосредственно предоставляющим любое свое содержимое в качестве внедряемого значения.

Определите маркер, указав однозначную символьную строку в его конструкторе следующим образом:

```
[app/logo-url.token.ts]

import {OpaqueToken} from '@angular/core';

export const LOGO_URL = new OpaqueToken('logo.url');
```

Внедрите этот маркер в определение модуля приложения обычным образом. Но при этом необходимо обозначить, на что именно он будет указывать после сопоставления. В данном случае он должен указывать на символьную строку изображения в URL:

```
[app/app.module.ts]

import {NgModule} from '@angular/core';
import {BrowserModule} from '@angular/platform-browser';
import {RootComponent} from './root.component';
import {ArticleComponent} from './article.component';
import {LOGO_URL} from './logo-url.token';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    RootComponent,
    ArticleComponent
  ],
  providers: [
    {provide: LOGO_URL, useValue:
              'https://angular.io/resources/images/
              ↳logos/standard/logo-nav.png'}
  ],
  bootstrap: [
    RootComponent
  ]
})
export class AppModule {}
```

И, наконец, данный маркер можно внедрить в компонент. Но поскольку внедряется нечто, еще не определенное с помощью декорации `@Injectable()`, то в конструкторе следует использовать декорацию `@Inject()`, чтобы сообщить Angular, что это должно быть сделано путем внедрения зависимостей. Кроме того, внедрение не присоединяется к ссылке `this` компонента автоматически, а следовательно, это придется также сделать вручную, как показано ниже. В конечном итоге изображение должно быть благополучно отображено в окне браузера!

```
[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {LOGO_URL} from './logo-url.token';

@Component({
  selector: 'article',
  template: `
    <img src="{{logoUrl}}">
    <h2>Fool and His Money Reunited at Last</h2>
    <p>Author: Jake Hsu</p>
  `
})
export class ArticleComponent {
  logoUrl:string;
  constructor(@Inject(LOGO_URL) private logoUrl_) {
    this.logoUrl = logoUrl_;
  }
}
```

Принцип действия

Класс `OpaqueToken` позволяет пользоваться неклассовыми типами в схеме поставщиков, ориентированной на классы и внедренной в Angular 2. Именно этот класс будет использоваться в каркасе внедрения зависимостей при попытке сопоставить маркеры внедрения зависимостей с объявлениями поставщиков. Это дает возможность более широко употреблять внедрение зависимостей повсюду в приложении, поскольку значение любого типа можно теперь предоставить везде, где бы ни внедрялась служба определенного типа.

Дополнение

Другой удобный способ внедрения значений дает возможность имитации службы. Допустим, требуется определить стандартную службу-заглушку, которую следует затем переопределить с помощью явного поставщика, чтобы активизировать полезное поведение. В подобных случаях нетрудно представить себе стандартный элемент статьи, который можно было бы по-другому сконфигурировать через директиву, повторно используя тот же самый компонент. Таким образом, в определенном ниже компоненте типа `ArticleComponent` будет использоваться имитирующая служба, когда директива не присоединяется, а когда она присоединяется, то конкретная служба.

```
[app/root.component.ts]

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

```
@Component({
  selector: 'root',
  template:
    <article></article>
    <article editor-view></article>
  `
})
export class RootComponent {}

[app/editor-article.service.ts]

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

export const MockEditorArticleService = {
  getArticle: () => ({
    title: "Mock title",
    body: "Mock body"
  })
};

@Injectable()
export class EditorArticleService {
  private title_:string =
      "Prominent Vegan Embroiled in Scrambled Eggs Scandal";
  private body_:string =
      "Tofu Farming Alliance retracted their endorsement.";

  getArticle() {
    return {
      title: this.title_,
      body: this.body_
    };
  }
}

[app/editor-view.directive.ts]

import {Directive} from '@angular/core';
import {EditorArticleService} from './editor-article.service';

@Directive({
  selector: '[editor-view]',
   providers: [EditorArticleService]
})
export class EditorViewDirective {}

[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {EditorArticleService} from './editor-article.service';
```

```
@Component({
  selector: 'article',
  template: `
    <h2>{{title}}</h2>
    <p>{{body}}</p>
  `
})
export class ArticleComponent {
  title:string;
  body:string;
  constructor(
        private editorArticleService_:EditorArticleService) {
    let article = editorArticleService_.getArticle();
    this.title = article.title;
    this.body = article.body;
  }
}
```

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`" с обширным обзором, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`", в котором демонстрируется, как перехватывать запросы от поставщиков внедрения зависимостей.
- Рецепт "Создание настраиваемой поставщиком службы с помощью свойства `useFactory`" с подробным описанием процесса установки фабрики служб для формирования определений настраиваемых служб.

Создание настраиваемой поставщиком службы с помощью свойства `useFactory`

Еще одним расширением внедрения зависимостей в Angular 2 является возможность пользоваться фабриками при определении иерархии поставщиков. Фабрика поставщиков позволяет принимать входные данные, выполнять произвольные операции, чтобы сконфигурировать поставщика и возвратить экземпляр этого поставщика для внедрения.

Исходный код, ссылки и практические примеры, связанные с данным рецептом, доступны по ссылке **http://ngcookbook.herokuapp.com/0049/**.

Подготовка

Начните снова с установки компонента, состоящего из двойной службы и статьи и представленного ранее в рецепте "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`".

Реализация

Фабрики поставщиков в Angular 2, как подразумевает их название, являются функциями, возвращающими поставщика. Фабрика может быть указана в отдельном файле и доступна по ссылке в свойстве `useFactory`.

Начните с объединения двух служб в единую службу, для конфигурирования которой должен быть вызвав соответствующий метод, как показано ниже.

```
[app/article.service.ts]

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

@Injectable()
export class ArticleService {
  private title_:string =
          "Flying Spaghetti Monster Sighted";
  private body_:string =
          "Adherents insist we are missing the point";
  private notes_:string = "Spot on!";
  private editorEnabled_:boolean = false;

  getArticle():Object {
    var article = {
      title: this.title_,
      body: this.body_
    };
    if (this.editorEnabled_) {
      Object.assign(article, article, {
        notes: this.notes_
      });
    }
    return article;
  }

  enableEditor():void {
    this.editorEnabled_ = true;
  }
}
```

Определение фабрики

Цель состоит в том, чтобы сконфигурировать данную службу таким образом, чтобы вызывать метод `enableEditor()`, исходя из состояния логического признака. Это можно сделать с помощью фабрик поставщиков. Поэтому определите фабрику в отдельном файле следующим образом:

```
[app/article.factory.ts]

import {ArticleService} from './article.service';

export function articleFactory(
       enableEditor?:boolean):ArticleService {
  return (articleService:ArticleService) => {
    if (enableEditor) {
      articleService.enableEditor();
    }
    return articleService;
  }
}
```

Внедрение маркера типа `OpaqueToken`

Превосходно! А далее необходимо переконфигурировать компонент `ArticleComponent`, чтобы внедрить маркер вместо требующейся службы:

```
[app/article.token.ts]

import {OpaqueToken} from '@angular/core';

export const ArticleToken = new OpaqueToken('app.article');

[app/article.component.ts]

import {Component, Inject} from '@angular/core';
import {ArticleToken} from './article.token';

@Component({
  selector: 'article',
  template: `
    <h2>{{article.title}}</h2>
    <p>{{article.body}}</p>
    <p *ngIf="article.notes">
      <i>Notes: {{article.notes}}</i>
    </p>
  `
})
export class ArticleComponent {
  article:Object;
  constructor(@Inject(ArticleToken) private articleService_) {
```

```
    this.article = articleService_.getArticle();
  }
}
```

Создание директив поставщиков с помощью свойства useFactory

И, наконец, необходимо определить директивы, чтобы указать порядок применения данной фабрики и внедрения этих директив в приложение, как показано ниже. Таким образом, должны быть обнаружены обе версии компонента ArticleComponent.

```
[app/default-view.directive.ts]

import {Directive} from '@angular/core';
import {ArticleService} from './article.service';
import {articleFactory} from './article.factory';
import {ArticleToken} from './article.token';

@Directive({
  selector: '[default-view]',
  providers: [
    {provide: ArticleToken,
      useFactory: articleFactory(),
      deps: [ArticleService]
    }
  ]
})
export class DefaultViewDirective {}

[app/editor-view.directive.ts]

import {Directive} from '@angular/core';
import {ArticleService} from './article.service';
import {articleFactory} from './article.factory';
import {ArticleToken} from './article.token';

@Directive({
  selector: '[editor-view]',
  providers: [
    {
      provide: ArticleToken,
      useFactory: articleFactory(true),
      deps: [ArticleService]
    }
  ]
})
export class EditorViewDirective {}
```

```
[app/root.component.ts]

import {Component} from '@angular/core';

@Component({
  selector: 'root',
  template: `
    <article default-view></article>
    <article editor-view></article>
  `
})
export class RootComponent {}
```

Принцип действия

Компонент статьи переопределяется таким образом, чтобы использовать маркер вместо внедрения службы. При наличии маркера Angular может обойти иерархическое дерево компонентов, чтобы обнаружить место, где предоставляется этот маркер. В директивах объявляется, что маркер сопоставляется с фабрикой поставщиков, т.е. с методом, вызываемым для возврата конкретного поставщика. Именно свойство `useFactory` и сопоставляет маркер с фабричным методом, а свойство `deps` — с зависимостями служб, которыми обладает фабрика.

Дополнение

Важное отличие, которое следует признать в данный момент, заключается в том, что все эти конфигурации фабрик происходят прежде получения экземпляров любых компонентов. После установки в декорации класса, где определяются поставщики, будет вызван фабричный метод.

См. также

- Рецепт "Контроль над созданием экземпляра службы и внедрением средствами класса `NgModule`", в котором дается обширный обзор, каким образом в Angular 2 строятся иерархии поставщиков с помощью модулей.
- Рецепт "Назначение псевдонимов при внедрении служб с помощью свойств `useClass` и `useExisting`", в котором показано, как перехватывать запросы от поставщиков внедрения зависимостей.
- Рецепт "Внедрение значения в виде службы с помощью свойства `useValue` и класса `OpaqueToken`", где показывается, как пользоваться маркерами внедряемых зависимостей для внедрения обобщенных объектов.