

Advanced Web Technology Lab 3

概述

Lab 3 的主要任务包括：

- 1.学习前端框架 (Angular)
- 2.学习后端框架 (Spring Boot)

Part 1：前端学习

1.安装Node.js与npm

首先，如果你的电脑里没有 Node.js 和 npm，请安装它们。

完整安装教程参考：https://blog.csdn.net/qq_42006801/article/details/124830995

Node.js

Node.js 是一个基于 Chrome V8 引擎的 JavaScript 运行时，可以近似理解为脱离浏览器运行 JavaScript 代码的平台。常见用途是开发后端服务器，但我们期末需要提交的 PJ 中建议使用Spring boot 开发后端，而不是Node.js。

建议去[官网](#)下载安装

npm

npm = Node Package Manager，是绝大部分现代前端项目中使用的包管理工具，会和Node.js一起安装。

由于npm的服务器在海外，所以访问速度比较慢，访问不稳定。推荐使用阿里的国内 npm 镜像：<https://registry.npmmirror.com> 并安装npm的镜像cnpm。

但是cnpm一般只用于模块安装，在项目创建与卸载等相关操作时仍使用npm。

在终端/控制台窗口中运行命令 `node -v` 和 `npm -v` 检查安装情况

```
C:\Users\Lenovo>node -v
v18.16.0

C:\Users\Lenovo>npm -v
6.14.18

C:\Users\Lenovo>
```

2.安装Angular

Angular 是一个基于 [TypeScript](#) 构建的开发平台。它包括：

- 一个基于组件的框架，用于构建可伸缩的 Web 应用
- 一组完美集成的库，涵盖各种功能，包括路由、表单管理、客户端-服务器通信等
- 一套开发工具，可帮助你开发、构建、测试和更新代码

本章的目标是使用 [Angular CLI 工具](#) 搭建你的 Angular 开发环境。包括：前提条件、安装 CLI、创建初始工作区和入门应用，以及在本地运行这个应用来验证你的搭建成果。详细内容参考 [angular官网](#)

安装 Angular CLI

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

在终端/控制台窗口中运行命令 `ng version` 检查安装情况

```
C:\Users\Lenovo>ng version

Angular CLI
Angular CLI: 15.2.6
Node: 18.16.0
Package Manager: npm 6.14.18
OS: win32 x64

Angular: undefined
...
Package      Version
-----
@angular-devkit/architect 0.1502.6 (cli-only)
@angular-devkit/core      15.2.6 (cli-only)
@angular-devkit/schematics 15.2.6 (cli-only)
@angular/cdk              15.2.8
@angular/material         15.2.8
@schematics/angular       15.2.6 (cli-only)
```

创建工作区和初始应用

Angular工作区就是你开发应用的上下文环境。每个工作区包含一些供一个或多个项目使用的文件。每个项目都是一组由应用、库或端到端（e2e）测试构成的文件。要想创建工作区和初始应用项目：

运行 CLI 命令 `ng new` 生成一个新项目以及应用的骨架代码：

```
ng new angular-learning
```

`ng new` 会提示你要把哪些特性包含在初始的应用项目中。请按回车接受默认值。

Angular CLI会安装必要的Angular npm包及其它依赖。这可能要花几分钟。

安装失败的同学可以尝试利用cnpm安装依赖包

```
ng new angular-learning --skip-install
cd angular-learning
cnpm install
```

还将创建下列工作区和初始项目文件：

- 一个新的工作区，根目录名叫 `angular-learning`
- 一个初始的骨架应用项目，也叫 `angular-learning`（但位于 `src` 子目录下）
- 一个端到端测试项目（位于 `e2e` 子目录下）
- 相关的配置文件

初始的应用项目是一个简单的“欢迎”应用，随时可以运行它。

启动开发服务器

Angular包含一个开发服务器，以便能轻易地在本地构建应用和启动开发服务器。

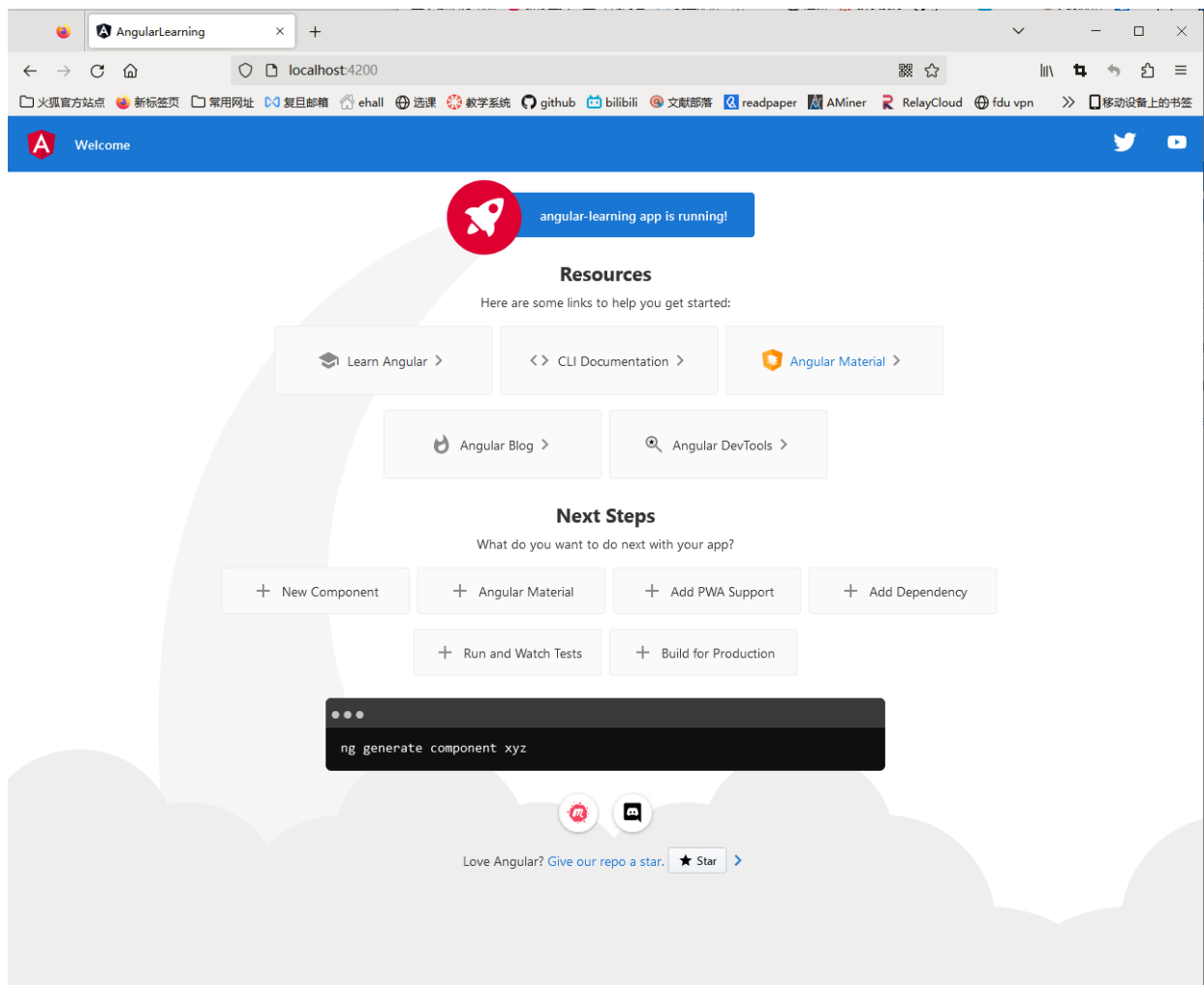
1. 进入项目目录
2. 使用CLI命令 `ng server` 启动开发服务器

```
cd angular-learning  
ng serve --open
```

`ng serve` 命令会启动开发服务器，监听文件变化，并在修改这些文件时重新构建此应用。

使用 `--open`（或 `-o`）参数可以自动打开浏览器并访问 <http://localhost:4200/>。

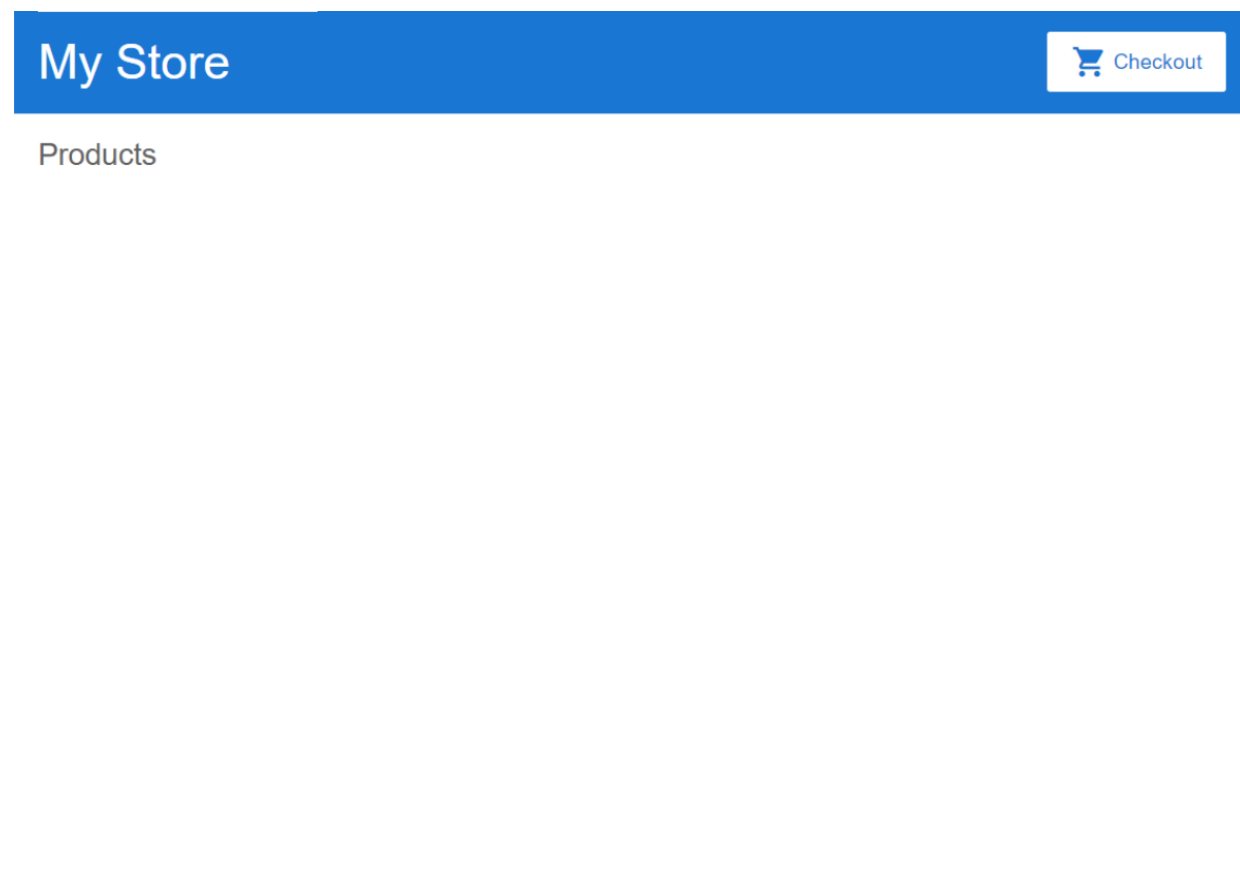
会看到：



3. Angular 起步

在这一部分，我们会先介绍Angular的基本概念，并通过构建一个电子商务网站，向你介绍 Angular 的基本知识

请同学们先下载[项目demo](#)。下载之后先尝试用 `npm install` 和 `ng serve` 把网站的“壳”运行起来。



可以看到，在这个初始应用程序中，定义了一个带有带有顶栏的框架（包含商店名称和结账图标）以及一个产品列表的标题（它将用来自应用中的数据填充并动态更新产品列表）。

Angular 基本概念

推荐同学们先把这两个文档看一下，建立一个大概的认识。

Angular项目代码结构：[工作区与项目文件的结构](#)

Angular的基本概念：[基本概念简介](#)

TypeScript

Angular 项目一般使用 TypeScript 来代替 JavaScript。直接写 JavaScript 是合法的，但是不推荐。

TypeScript 是一种编译到 JavaScript 的编程语言，弥补了一些 JavaScript 语言上的一些缺点，比 JavaScript 更加强大好用。

TypeScript学习：<https://www.typescriptlang.org/index.html>

理解起始工程

下面让我们来理解一下工程的起始页面是如何工作的。

- `src/app/app.module.ts` 的主要作用是导入需要用到的库（如 `BrowserModule`），并声明项目中包含的组件（如 `AppComponent`）。
- `src/app/app.component.ts` 中定义了名为 `AppModule` 的根模块，它会告诉 Angular 如何组装应用。这里最初只声明一个 `AppComponent`。当你向应用中添加更多组件时，它们也必须在这里声明。
- `selector: 'app-root'` 对应的是 `index.html` 中的 `<app-root></app-root>`，表示当前组件的代码会被插入到 `<my-app>` 标签中。
- 在 `src/app` 路径下已经生成了两个组件 `top-bar` 和 `product-list`。

模板语法

Angular 的模板语法扩展了 HTML 和 JavaScript。在接下来的步骤中，你将了解 Angular 的五个常用模板语法

- `*ngFor`
- `*ngIf`
- 插值 `{{}}`
- 属性绑定 `[]`
- 事件绑定 `()`

1. 在 `product-list` 文件夹中，打开模板文件 `product-list.component.html`。
2. 在 `<div>` 上添加一个结构型指令 `*ngFor`，如下所示。

```
<h2>Products</h2>

<div *ngFor="let product of products">
</div>
```

`*ngFor` 是一个 "结构型指令"。结构型指令会通过添加、删除和操纵它们的宿主元素等方式塑造或重塑 DOM 的结构。

任何带有星号 `*` 的指令都是结构型指令。

3. 在此 `<div>` 中，添加 `<h3>` 和 `{{ product.name }}`。

```
<h2>Products</h2>

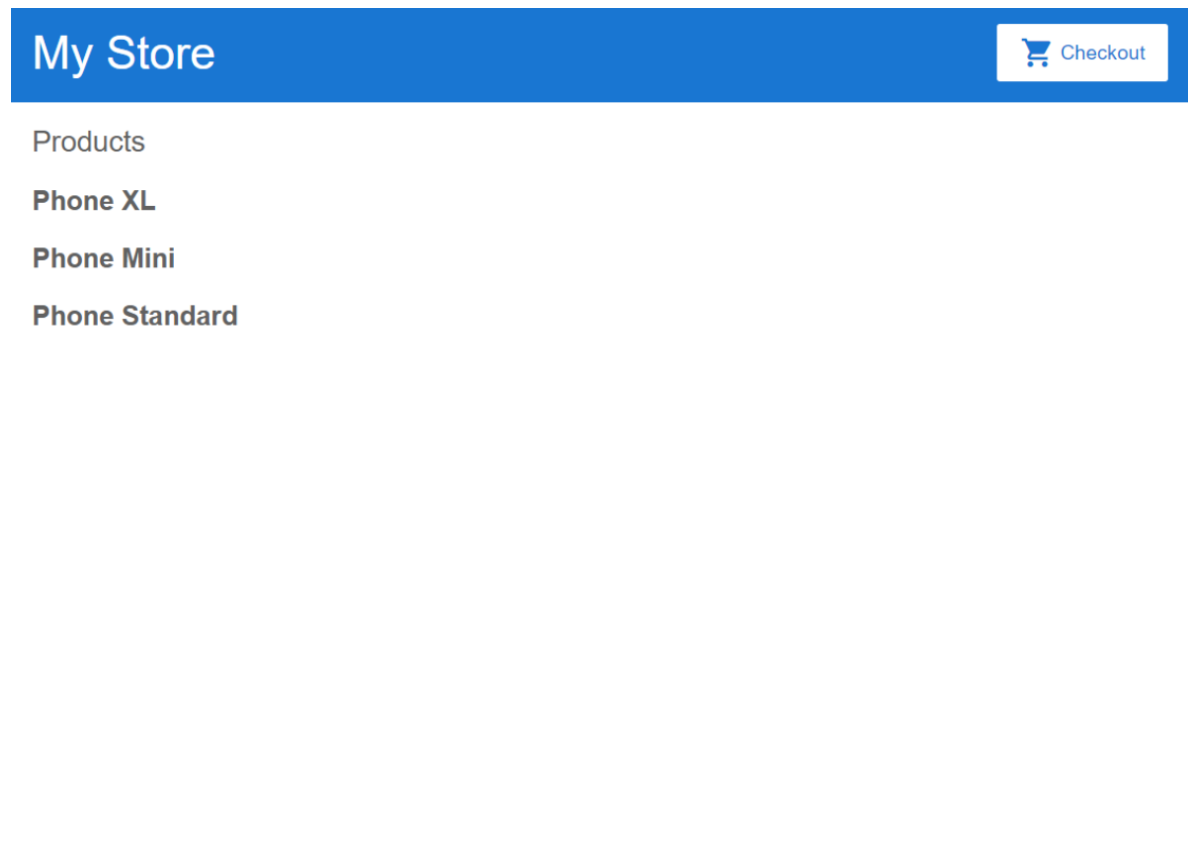
<div *ngFor="let product of products">

  <h3>
    {{ product.name }}
  </h3>

</div>
```

插值语法 `{{ }}` 可以让你把属性值渲染为文本。

预览窗格会立即更新，以显示列表中每个商品的名称。



4. 为了让每个商品名称都能链接到商品详情，添加一个 `<a>` 元素，并使用属性绑定语法 `[]` 把该链接的 `title` 设置成该商品的名字，如下所示：

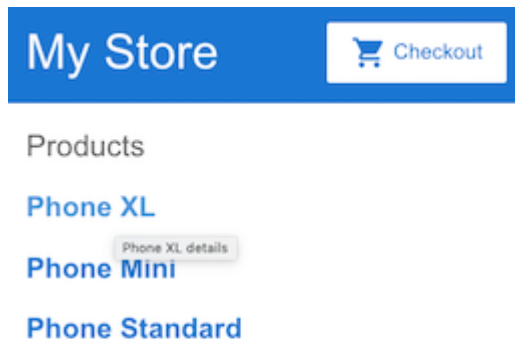
```
<h2>Products</h2>

<div *ngFor="let product of products">

  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

</div>
```

在预览窗格中，将鼠标悬停在显示的商品名称上，可以看到绑定的 `name` 属性值。它们都是商品名加上单词 "details" 的格式。



插值 `{{}}` 允许你把属性值渲染为文本；而属性绑定语法 `[]` 则允许你在模板表达式中使用属性值。

5. 添加商品说明。在 `<p>` 标签上使用 `*ngIf` 指令，这样 Angular 只让当前产品有描述 `<p>`

```
<h2>Products</h2>

<div *ngFor="let product of products">

  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

  <p *ngIf="product.description">
    Description: {{ product.description }}
  </p>

</div>
```

该应用会立即在列表中显示每种商品的名称和描述。请注意，最后一个商品根本没有描述信息。由于该商品的 `description` 属性为空，因此 Angular 不会创建 `<p>` 元素（包括静态文本“Description”）。

Products

Phone XL

Description: A large phone with one of the best screens

Phone Mini

Description: A great phone with one of the best cameras

Phone Standard

6. 添加一个按钮，以便用户可以共享产品。将按钮的 `click` 事件绑定到 `product-list.component.ts` 中的 `share()` 方法。事件绑定要在此事件用一组圆括号 `()` 括起来，就比如 `<button>` 元素上的 `(click)`。

```
<h2>Products</h2>

<div *ngFor="let product of products">

  <h3>
    <a [title]="product.name + ' details'">
      {{ product.name }}
    </a>
  </h3>

  <p *ngIf="product.description">
    Description: {{ product.description }}
  </p>

  <button type="button" (click)="share()">
    share
  </button>

</div>
```

现在，每个商品都有一个“Share”按钮了：

Products

Phone XL

Description: A large phone with one of the best screens

[Share](#)

Phone Mini

Description: A great phone with one of the best cameras

[Share](#)

Phone Standard

[Share](#)

目前，该应用现在具有商品列表和共享功能。在这个过程中，你已经学会了 Angular 模板语法的五个常用特性：

- `*ngFor`
- `*ngIf`
- 插值 `{{}}`
- 属性绑定 `[]`
- 事件绑定 `()`

组件

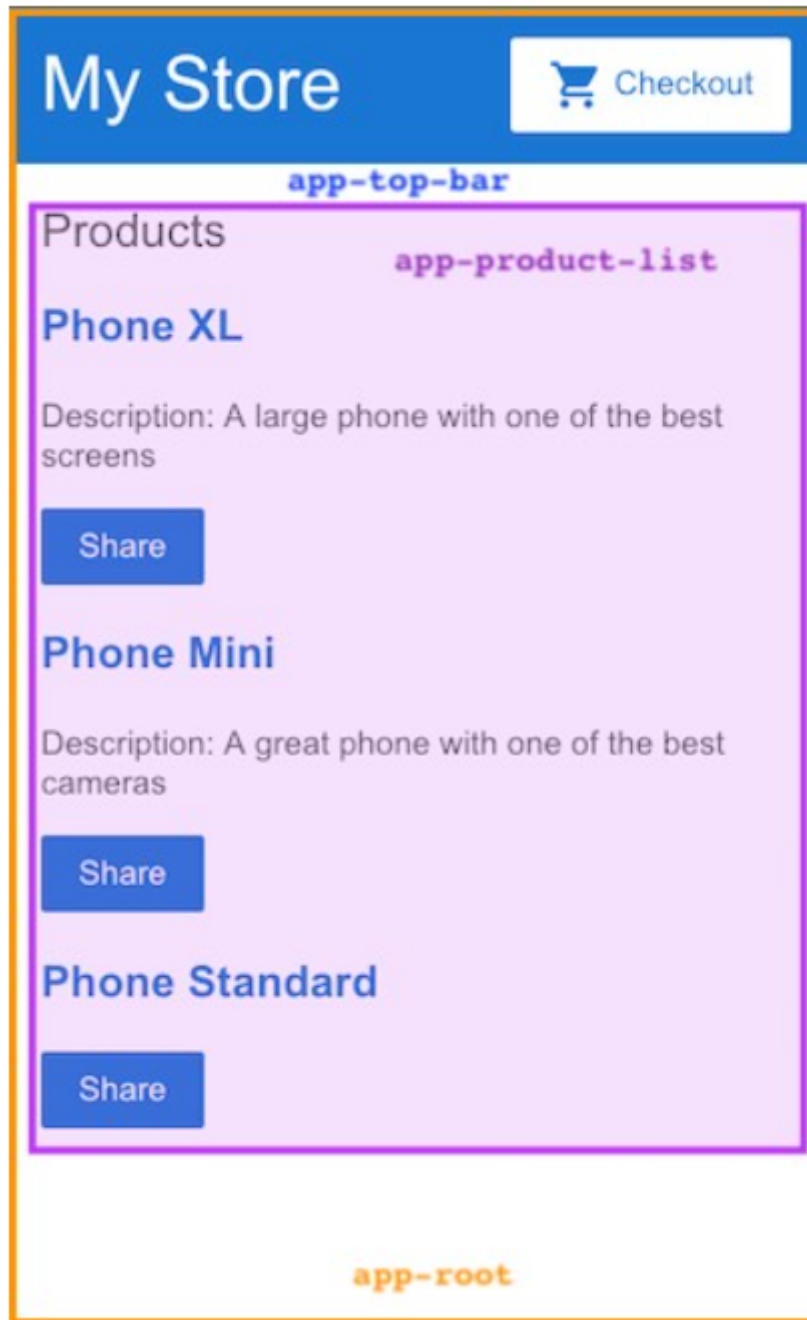
组件在用户界面（也就是 UI）中定义了一些责任区，让你能重用这些 UI 功能集。你

组件包含三部分：

- **一个组件类** 它用来处理数据和功能。上一节，我们在组件类中定义了商品数据和 `share()` 方法，它们分别用来处理数据和功能。
- **一个 HTML 模板** 它决定了 UI。在上一节中，商品列表的 HTML 模板用来显示每个商品的名称、描述和“Share”按钮。
- **组件专属的样式** 定义了外观和感觉。商品列表中还没有定义任何样式，那属于组件 CSS 负责。

Angular 应用程序由一棵组件树组成，每个 Angular 组件都有一个明确的用途和责任。

目前，该范例有三个组件：



- `app-root`（橙色框）是应用的外壳。这是要加载的第一个组件，也是所有其它组件的父组件。你可以把它想象成一个基础页面。
- `app-top-bar`（蓝色背景）是商店名称和结帐按钮。
- `app-product-list`（紫色框）是在上一节中修改过的商品列表。

组件交互

目前，产品列表中显示了每个产品的名称和描述。`ProductListComponent` 还定义了一个 `products` 属性，包含从 `products.ts` 的 `products` 数组导入的各个产品的数据。

下一步是创建一个新的通知功能，该功能会使用来自 `ProductListComponent` 的产品数据。通知会检查产品的价格，如果价格大于 700 美元，则会显示 **Notify Me** 按钮，当产品上市销售时，用户可以通过该按钮注册通知。

将数据传递到子组件

本节将引导你创建一个子组件 `ProductAlertsComponent`，该子组件可以从其父组件 `ProductListComponent` 接收数据。

1. 终端运行以下命令生成一个名为 `product-alerts` 的新组件。

```
ng generate component product-alerts
```

该 generator 为组件的三个部分创建了启动文件，并自动把 `ProductAlertsComponent` 添加到 `AppModule` 中，以便它能用于本应用的其它组件中。

- `product-alerts.component.ts`
- `product-alerts.component.html`
- `product-alerts.component.css`

2. 打开 `product-alerts.component.ts`。

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

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

  ngOnInit() {
  }
}
```

`@Component()` 是个装饰器函数，用于为该组件指定 Angular 所需的元数据,主要功能如下：

- `selector` (`app-product-alerts`) 用于标识组件。按照惯例，Angular 组件选择器以前缀 `app-` 开头，后跟组件名称。
- 模板文件名 `templateUrl` 和样式文件名 `styleUrls` 引用了组件的 HTML 和 CSS。
- 这个 `@Component()` 定义还导出了类 `ProductAlertsComponent`，该类会处理组件的功能。

`ngOnInit()` 是一个[生命周期钩子](#)，Angular 在创建完组件后很快就会调用 `ngOnInit()`。这里是放置初始化逻辑的好地方。始终要 `export` 这个组件类，以便在其它地方（比如 `AppModule`）导入它。

3. 要将 `ProductAlertsComponent` 设置为接收产品数据，请首先从 `@angular/core` 中导入符号 `Input`。然后在 `ProductAlertsComponent` 类定义中，使用 `@Input()` 装饰器定义一个名为 `product` 的属性。

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

export class ProductAlertsComponent implements OnInit {
  @Input() product;
  constructor() { }

  ngOnInit() {
  }
}
```

4. `@Input()` 装饰器指出其属性值是从该组件的父组件商品列表组件中传入的(`product-alerts.component.ts`)。

5. 定义这个新商品提醒组件的视图：

打开 `product-alerts.component.html` 并将占位符段落替换为 **Notify Me** 按钮，如果产品价格超过 700 美元，就会出现此按钮。

```
<p *ngIf="product && product.price > 700">
  <button type="button">Notify Me</button>
</p>
```

6. 最后，要将 `ProductAlertsComponent` 显示为 `ProductListComponent` 的子级，请将 `<app-product-alerts>` 元素添加到 `product-list.component.html` 中。使用属性绑定将当前产品作为输入传给此组件。

```
<button type="button" (click)="share()">
  Share
</button>

<app-product-alerts
  [product]="product">
</app-product-alerts>
```

7. 新商品提醒组件会从商品列表中获取商品作为输入信息。通过该输入，它会根据商品的价格显示或隐藏“Notify Me”按钮。由于 Phone XL 的售价超过了 700 美元，所以该商品上会出现“Notify Me”按钮。

Products

Phone XL

Description: A large phone with one of the best screens

Share

Notify Me

Phone Mini

Description: A great phone with one of the best cameras

Share

Phone Standard

Share

要了解如何将数据从父组件传递到子组件、拦截并处理来自父组件的值，以及检测并对输入属性值更改的更多信息，请参阅 [组件交互](#) 一章。

将数据传递到父组件

为了使 **Notify Me** 按钮起作用，子组件需要通知并将数据传递给父组件。当用户单击 **Notify Me** 时 `ProductAlertsComponent` 需要引发一个事件，并且 `ProductListComponent` 需要响应此事件。

1. 打开 `product-alerts.component.ts`，从 `@angular/core` 中导入 `Output` 和 `EventEmitter`：

```
import { Component } from '@angular/core';
import { Input } from '@angular/core';
import { Output, EventEmitter } from '@angular/core'; //新增行
```

2. 在组件类中，添加一个用 `@Output()` 装饰器和一个事件发射器 `EventEmitter()` 实例定义一个名为 `notify` 的属性。

```
export class ProductAlertsComponent implements OnInit {
  @Input() product;
  @Output() notify = new EventEmitter();
  constructor() { }

  ngOnInit() {
  }
}
```

使用 `@Output()` 配置 `ProductAlertsComponent`，这会让 `ProductAlertsComponent` 在 `notify` 属性的值发生变化时引发一个事件。

3. 在 `product-alerts.component.html` 中, 修改 **Notify Me** 按钮, 增加事件绑定, 并调用 `notify.emit()` 方法。

```
<p *ngIf="product && product.price > 700">
  <button type="button" (click)="notify.emit()">Notify Me</button>
</p>
```

4. 定义用户单击按钮时发生的行为。当子组件引发事件时, 父组件 `ProductListComponent` (而不是 `ProductAlertsComponent`) 会采取行动。在 `product-list.component.ts` 中, 定义一个 `onNotify()` 方法, 类似于 `share()` 方法。

```
export class ProductListComponent {
  products = products;

  share() {
    window.alert('The product has been shared!');
  }

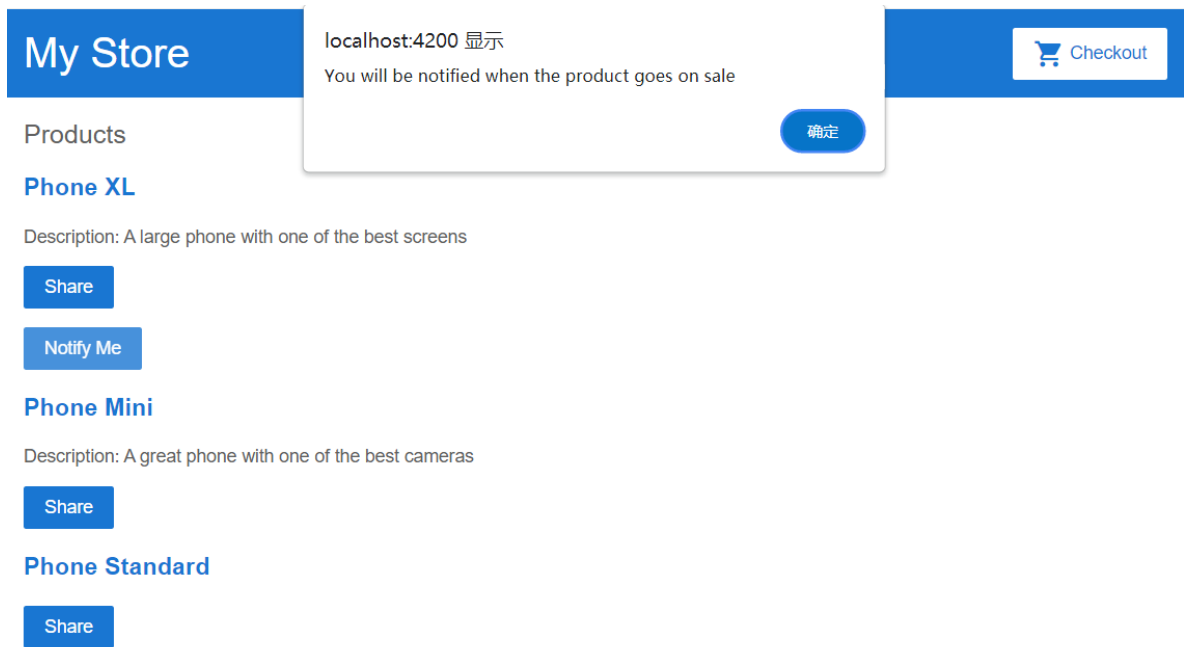
  onNotify() {
    window.alert('You will be notified when the product goes on sale');
  }
}
```

5. 更新 `ProductListComponent`, 以从 `ProductAlertsComponent` 中接收数据。

在 `product-list.component.html` 中, 将 `<app-product-alerts>` 绑定到产品列表组件的 `onNotify()` 方法。 `<app-product-alerts>` 会显示 **Notify Me** 按钮的内容。

```
<button type="button" (click)="share()">
  Share
</button>

<app-product-alerts
  [product]="product"
  (notify)="onNotify()">
</app-product-alerts>
```



要了解关于从子组件监听事件、读取子属性或调用子方法以及如何用服务在组件之间进行双向通信的详细信息，请参阅[“组件交互”](#)一章。

路由

现在，这个在线商店应用会有一个基本的商品名录。该应用还没有任何可变的州或导航。它只有一个 URL，该 URL 总是会显示“我的商店”页面，其中是商品列表及其描述。

接下来我们尝试添加 Angular 路由器，来用一些独立页面显示完整的产品详情，这些页面有自己的 URL。

Angular [路由器](#)能让你根据用户在应用中的位置向用户显示不同的组件和数据。当用户执行应用任务时，路由器可以从一个视图导航到另一个视图。比如：

- 在地址栏中输入一个 URL，导航到相应的页面。
- 点击页面上的链接，导航到新页面。
- 点击浏览器的后退和前进按钮，在浏览器的历史中前后导航。

关联 URL 路径与组件

1. 生成用于展示产品详情的新组件。在终端窗口运行如下命令来生成一个新的 `product-details` 组件

```
ng generate component product-details
```

2. 在 `app.module.ts` 中，添加一个商品详情路由规则，该路由的 `path` 是 `products/:productId`，`component` 是 `ProductDetailsComponent`。

```
@NgModule({
  imports: [
    BrowserModule,
    ReactiveFormsModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
    ])
  ],
  declarations: [
    AppComponent,
    TopBarComponent,
    ProductListComponent,
    ProductAlertsComponent,
    ProductDetailsComponent,
  ],

```

典型的 Angular `Route` 具有两个属性：

- `path`: 用来匹配浏览器地址栏中 URL 的字符串。
- `component`: 导航到该路由时，路由器应该创建的组件。

`@NgModule` 元数据会初始化路由器，并开始监听浏览器地址的变化。

将 `RouterModule` 添加到 `imports` 数组中，同时通过调用 `RouterModule.forRoot()` 来使用路由配置：这个方法之所以叫 `forRoot()`，是因为你要在应用的顶级配置这个路由器。`forRoot()` 方法会提供路由所需的服务提供商和指令，还会基于浏览器的当前 URL 执行首次导航。

3. 配置组件的模板，以定义用户如何导航到路由或 URL。当用户点击商品名称时，应用就会显示那个商品的详情。

打开 `product-list.component.html`。修改 `*ngFor` 指令，在遍历列表的过程中把 `products` 数组中的每个索引赋值给 `productId` 变量。修改商品名称的链接，使其包含 `routerLink`。

```
<div *ngFor="let product of products">

  <h3>
    <a
      [title]="product.name + ' details'"
      [routerLink]="['/products', product.id]">
        {{ product.name }}
      </a>
    </h3>

    <!-- . . . -->

</div>
```

`RouterLink` 指令让路由器控制了一个链接元素。在这种情况下，路由或 URL 包含一个固定的区段（`/products`），但其最后一个区段是变量，要插入当前商品的 id 属性。例如，id 为 1 的商品的 URL 类似于 `http://localhost:4200/products/1`。

4. 通过单击商品名称来测试路由器。该应用会显示商品详情组件，该组件目前始终显示 “product-details works! ”



注意窗口中的 URL 变化了。它的最后一段是 `products/#`，这里的 `#` 代表你点击的那个路由的编号。

使用路由信息

商品详情组件负责处理每个商品的显示。Angular 的路由器会根据浏览器的 URL 和你定义的这些路由来决定如何显示组件。

本节通过 Angular 的路由器来组合使用 `products` 数据和路由信息，以显示每个商品的详情。

1. 打开 `product-details.component.ts`，从 `@angular/router` 导入 `ActivatedRoute`，并从 `../products` 导入 `products` 数组。

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

import { products } from '../products';
```

2. 定义 `product` 属性，并通过把 `private route: ActivatedRoute` 添加为构造函数括号内的参数，将 `ActivatedRoute` 注入到 `constructor()` 中。

```
export class ProductDetailsComponent implements OnInit {
  product;

  constructor(private route: ActivatedRoute,) { }

  ngOnInit() {}
}
```

`ActivatedRoute` 专门用于由 Angular 路由器加载的每个路由组件。它包含关于该路由，路由参数以及与该路由关联的其它数据的信息。

3. 在 `ngOnInit()` 方法中订阅路由参数，并且根据 `productId` 获取该产品。

```
ngOnInit() {
  this.route.paramMap.subscribe(params => {
    this.product = products[+params.get('productId')];
  });
}
```

这个路由参数对应于你在路由中定义的路径变量。与该路由匹配的 URL 提供了 `productId`。Angular 使用这个 `productId` 来显示每个单独商品的详细信息。

4. 更新 `ProductDetailsComponent` 的模板以显示带有 `*ngIf` 的产品详情。如果产品存在，则此 `<div>` 会显示名称、价格和说明。

```
<h2>Product Details</h2>

<div *ngIf="product">
  <h3>{{ product.name }}</h3>
  <h4>{{ product.price | currency }}</h4>
  <p>{{ product.description }}</p>
</div>
```

`<h4>{{ product.price | currency }}</h4>` 这一行，使用 `currency` 管道将 `product.price` 从数字转换为货币字符串。

管道是一种可以在 HTML 模板中转换数据的方式。有关 Angular 管道的更多信息，请参见[管道](#)。

现在，当用户点击商品列表中的某个名字时，路由器就会导航到商品的不同网址，用商品详情组件代替商品列表组件，并显示商品详情。

Product Details

Phone XL

\$799.00

A large phone with one of the best screens

4. angular+material UI 页面构建

Material UI 是一款功能非常强大，界面却十分清新简洁的CSS框架，Material UI利用了Google的Material Design 全新设计语言，并且让每一个UI组件都变得非常独立，因此开发者使用Material UI也会比较简单。和Bootstrap类似，Material UI提供了很多常用的UI组件，包括菜单、按钮、滑动杆、进度条、单选框/复选框等等。

本节我们将借助material UI的列表控件 ([Lists](#)) 来优化界面，最终实现目标如下图所示

Products

Phone XL

description: A large phone with one of the best screens

[Notify Me](#)[Share](#)

Phone Mini

description: A great phone with one of the best cameras

[Share](#)

Phone Standard

[Share](#)

1. 终端运行 `npm install` 命令安装material UI的依赖包

```
npm install --save @angular/material @angular/cdk @angular/animations
```

安装后 可以看到 `package.json` 文件更新了 `dependencies`

```
"name": "angular-demo",
"version": "0.0.0",
"private": true,
"dependencies": {
  "@angular/animations": "8.2.14",
  "@angular/common": "8.2.14",
  "@angular/compiler": "8.2.14",
  "@angular/core": "8.2.14",
  "@angular/forms": "8.2.14",
  "@angular/material": "^7.3.7",
  "@angular/platform-browser": "8.2.14",
  "@angular/platform-browser-dynamic": "8.2.14",
  "@angular/router": "8.2.14",
```

2. 在 `app.module.ts` 中导入需要使用的material组件

```

import { MatToolbarModule } from '@angular/material/toolbar';
import { MatListModule } from '@angular/material/list';
import { MatButtonModule } from '@angular/material/button';
import { MatCardModule } from '@angular/material/card';

@NgModule({
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    ReactiveFormsModule,
    MatButtonModule, MatCardModule,
    MatToolbarModule, MatListModule,
    RouterModule.forRoot([
      { path: '', component: ProductListComponent },
      { path: 'products/:productId', component: ProductDetailsComponent },
    ])
  ],
  declarations: [...]
  bootstrap: [ AppComponent ]
})

```

3. 用list控件重写 product-list.component.html

```

<mat-list style="width: 80%;margin: 0 auto;">
  <h3 mat-subheader>Products</h3>
  <mat-list-item *ngFor="let product of products; index as productId">
    <h4 mat-line>
      <a [title]="product.name + ' details'" [routerLink]="['/products',
productId]">
        {{ product.name }}
      </a>
    </h4>
    <p mat-line *ngIf="product.description">
      description: {{ product.description }}
    </p>
    <app-product-alerts [product]="product" (notify)="onNotify()">
    </app-product-alerts>
    <button style="margin-left: 20px;" mat-raised-button color="primary"
(click)="share()">Share</button>
  </mat-list-item>
</mat-list>

```

`<mat-list>` 是一个容器组件，它包装并格式化一系列 `<mat-list-item>`。基本格式如下：

```

<mat-list>
  <mat-list-item>列表 行1</mat-list-item>
  <mat-list-item>列表 行2</mat-list-item>
  .....
  <mat-list-item>列表 行n</mat-list-item>
</mat-list>

```

4. 定义 product-list.component.css

```
.example-card {
  max-width: 400px;
}

.example-header-image {
  background-image:
url('https://material.angular.io/assets/img/examples/shiba1.jpg');
  background-size: cover;
}
```

更多内容查看[material文档](#)

5. 继续学习 Angular

继续学习 Angular 请阅读官方 Tutorial。

相关参考：

1. [动态组件加载器](#)
2. [继续完善电商网站](#)
3. [更多demo](#)
4. [基本概念](#)
5. [模板引用变量 \(#var\)](#)
6.  [Angular 4.x 动态创建组件](#)

Angular 官方网站: <https://angular.io>, 中文网站: <https://angular.cn>。推荐先阅读文档中的 Tutorial 部分实现官方样例, 再阅读 Guide 部分详细了解 Angular 的工作原理。

Angular 开发相关资源: <https://angular.io/resources>, 中文: <https://angular.cn/resources>。

其中, IDE推荐使用 vS Code 或者 WebStorm, Tooling 部分推荐学习 Angular CLI, UI Component 部分列举了目前主要的 UI 库, 跨平台开发部分推荐了解 Ionic 和 Electron。

TypeScript: <https://www.typescriptlang.org/index.html>

npm: <https://docs.npmjs.com/getting-started>

china npm: <https://npm.taobao.org/>

Node.js: <https://nodejs.org>

5. 其它前端框架

Vue: [Vue中文文档](#)

React: [React中文文档](#)

Part 2: 后端学习

1. Spring Boot

前后端分离的 Web 架构中，后端一般是一个提供 Restful 接口的服务器，为前端提供所需数据。

目前主流的 Restful 后端的语言与框架有：

- JavaScript，运行于 Node.js 环境，有 Express, Restify 等框架
- Java，有 Spring Boot 等框架
- Python，有 Flask, Django web.py 等框架
- Ruby，有 Sinatra 等框架
- PHP，有 Laravel 等框架

课程 Pj 建议 后端采用SpringBoot 框架，以熟悉业界典型的前后端分离架构，并且和讲解课程内容保持一致。本次 Lab 贴合课程内容，以 Spring Boot 为例，开发一个简单的 Restful 服务。

环境依赖

JDK 与 JRE

没有 Java 环境的电脑需要同学们去安装 JDK 与 JRE。

Maven

Maven 是 Java 的库管理工具，其功能与 npm 比较类似。

如果不安装 Maven，我们也可以通过手动将 Jar 包放到程序的 classpath 中来添加依赖文件，就像前端也可以手动下载 js 库并添加至页面中一样。但是这样的方式不利于管理依赖文件，因此推荐使用 Maven。

下载 Maven: <http://maven.apache.org/download.cgi> 或解压 Lab 目录中的 apache-maven-3.5.0-bin.zip

安装 Maven: <http://maven.apache.org/install.html>

IntelliJ IDEA

IDEA 全称 IntelliJ IDEA，是[java](#)编程语言的集成开发环境，是JetBrains公司产品，同学们可以通过教育认证获取正版软件

安装教程参考 https://blog.csdn.net/qg_43554335/article/details/121928344

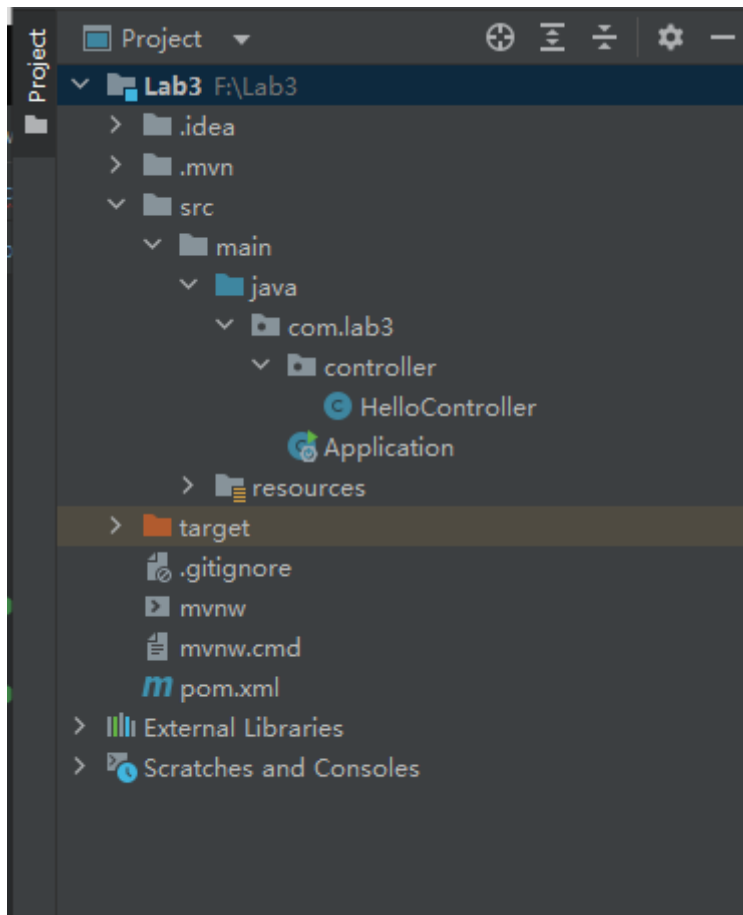
新建 Spring Boot 工程

新建 Spring Boot 工程参考官方教程: <https://spring.io/guides/gs/spring-boot/>。按此教程新建完工程后，在IntelliJ IDEA中打开要选择 Import Project 或者菜单栏中的 File - Project from Existing Source，并在其中选择 Maven 类型。

使用 IntelliJ IDEA Ultimate 版本的同学新建工程时也可以参考: <https://www.jetbrains.com/help/idea/creating-spring-boot-projects.html>

搭建完 Spring Boot 工程后，首先运行 `mvn install` 安装所需依赖。

初始目录结构 如下图所示



运行 Spring Boot

新建Spring Boot工程后, `Application.java` 为工程的入口, 运行此类便可运行我们的 Spring Boot 工程。代码如下:

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```

`SpringApplication.run(Application.class, args);` 一行运行了我们的 Spring 工程。Spring Boot 会搜索工程中的所有 Controller, 并注册对应的接口。然后 Spring Boot 会运行内置的 Tomcat 服务器来提供服务。

可以看到, 当前Spring Boot工程中已有一个controller接口: `controller/HelloController.java`


```
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;

@RestController
public class HelloController {
    @RequestMapping("/hello")
    public String index() {
        return "Hello, world!";
    }
}
```

- `@RestController` 这个 Annotation 标记了这个类为 Restful Service 的一个 Controller，这是每个 Controller 在类名前必须加的 Annotation。
- `@RequestMapping("/hello")` 表示 `index` 这个方法注册了 `/hello` 这个 path。可以通过在浏览器中打开 <http://localhost:8080/hello> 来访问这个接口。接口的 HTTP method 默认为 `GET`，其他 method 会在后面讲述。

终端运行命令 `mvnw spring-boot:run`，并在浏览器中打开 <http://localhost:8080/hello>，可以看到



注意，这个接口返回的只是一个 String，不是 JSON，所以这个接口并不是一个符合 Restful 标准的接口。在写代码时要避免出现这种不标准的情况。

创建第一个 Restful 接口

接下来我们创建一个简单的 Restful 接口。

接口 path 为 `/greeting`，接收一个 url 参数 `name`，方法为 `GET`。

样例 URL: <http://localhost:8080/greeting?name=Peter>。

样例返回:

```
{
  "id":1,
  "name":"Hello, Peter!"
}
```

实现方式如下:

1. 为返回的 JSON 对象创建对应的 Bean 类

新建 `response/GreetingResponse.java`，并添加如下代码:

```

public class GreetingResponse {
    private final long id;
    private final String name;

    public GreetingResponse(long id, String name) {
        this.id = id;
        this.name = name;
    }

    public long getId() {
        return id;
    }

    public String getName() {
        return name;
    }
}

```

这个类中，最主要的是 `getId` 和 `getName` 这两个 getter 方法，这两个方法决定了返回的 JSON 的样式。

Java Bean 和 JSON 直接的对应关系是由 `com.fasterxml.jackson.core` 这个包提供的，这个包被包含在了我们的 `spring-boot-starter-web` 中，如果是使用的 `spring-boot` 包，需要手动添加 `jackson` 包才能支持 Java Bean 和 JSON 之间的 Mapping。

具体包依赖关系在 `pom.xml` 中的 `dependencies` 中配置。

2. 创建接口函数

在 `HelloController.java` 将用到的类引用进来：

```

import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestParam;
import java.util.concurrent.atomic.AtomicLong;
import com.lab3.response.GreetingResponse;

```

并在 `HelloController` 类添加 `/greeting` 请求接口：

```

    private final AtomicLong counter = new AtomicLong();

    @CrossOrigin(origins = "*")
    @RequestMapping("/greeting")
    public @ResponseBody GreetingResponse greeting(@RequestParam(value = "name",
        defaultValue = "world") String name){
        return new GreetingResponse(counter.incrementAndGet(), "Hello, " + name
+ "!");
    }

```

- `counter` 是用来从 1 开始计数的，每次接口被调用时将 `counter` 加一。

- `@RequestMapping("/greeting")` 将本接口注册到 `/greeting` 上。
- `@ResponseBody GreetingResponse` 指定了接口返回的数据类型。
- 方法参数 `@RequestParam(value = "name", defaultValue = "world") String name` 中, 首先参数是 `String name`, `@RequestParam` 指定了这个参数是在 URL 的 path 上的一个参数, 而非在 header 中或者 body 中。`value = "name"` 指定了这个变量对应 path 上的 `name` 参数。`defaultValue = "world"` 表示如果没有 `name` 参数, 则将变量的值赋值为 `world`。

函数内部可以自由编写, 最后返回一个 `GreetingResponse` 对象即可。

终端运行命令 `mvnw spring-boot:run`, 并在浏览器中打开<http://localhost:8080/greeting?name=Peter>, 可以看到



支持跨域访问

目前我们的后端不能被前端直接调用, 这是因为我们还没有设置跨域访问的权限。

比如当我们在浏览器新的 Tab 页面的 console 中使用 ajax 调用后端服务, 会出现如下报错:

```
XMLHttpRequest cannot load http://localhost:8080/hello. No 'Access-Control-Allow-Origin' header is present on the requested resource. Origin '...' is therefore not allowed access.
```

为了让前端能够调用后端服务, 我们可以将前端域名添加到头部 `Access-Control-Allow-Origin` 属性中。

在接口函数前添加 `@CrossOrigin(origins = "<前端 hostname, 如 [http://localhost:9000]>")`, 或者方便起见允许来自所有源的请求, 如下所示:

```
@CrossOrigin(origins = "*")
@RequestMapping("/hello")
public String index() {
    return "Hello, world!";
}
```

Post请求方法

http/1.1协议中共定义了八种请求方式来表明request-url不同的操作, 分别是get、post、put、delete、head、options、trace、connect, 其中最常用的是前四种:

- Get: 请求指定的页面信息, 并返回实体主体
- Post: 向指定资源提交数据进行处理请求(例如提交表单或者上传文件)。数据被包含在请求体中。post请求可能会导致新的资源的建立和已有资源的修改。
- Put: 从客户端向服务器传送的数据取代指定的文档内容

- Delete: 请求服务器删除指定内容

对于同一个 `path`，可以注册多个方法不一样的接口。

之前我们的接口没有指定请求方法，默认请求方法为 **GET**。本节我们将通过在 `/register` 上的定义Post方法实现**用户注册**功能

1. 定义请求体

请求体样例：

```
{
  "username": "new-user",
  "password": "123456",
  "email": "a@b.com",
  "phone": "12345678900"
}
```

创建 `request/UserRegisterRequest.java`，并添加如下代码：

```
public class UserRegisterRequest {
    private String username;
    private String password;
    private String email;
    private String phone;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }
    // 省略剩下的 getter 与 setter
}
```

2. 定义响应体

标准的 Restful 服务返回的数据必须是 JSON，不能是一个字符串或者一个数字。

有时同一个接口可能返回不同结构的 JSON 数据，比如用户注册可能有注册成功和注册失败两种情况，会返回不同的数据。

创建 `response/UserRegisterResponse.java`，并添加如下代码：

```

public class UserRegisterResponse{
    private final String message;

    public UserRegisterResponse(String message) {
        this.message = message;
    }

    public String getMessage() {
        return message;
    }
}

```

同理，定义 `response/ErrorResponse.java`

3. 定义请求接口

创建 `controller/UserController.java`，并添加如下代码：

```

import com.lab3.request.UserRegisterRequest;
import com.lab3.response.ErrorResponse;
import com.lab3.response.UserRegisterResponse;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMethod;

@RestController
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public @ResponseBody Object register(@RequestBody UserRegisterRequest request){
        if (/* success */ true) {
            return new UserRegisterResponse("Register Successfully");
        } else {
            return new ErrorResponse("Failed to Register");
        }
    }
}

```

在类之前也可以加上 `@RequestMapping`。这样做的话，所有函数的 `@RequestMapping` 的 path 之前都会加上这个 path。比如上面代码中，注册的 path 是 `/user/register`。

常见请求参数

- `@RequestParam` 用来接收 path 中的变量，一个接口可以定义多个 `@RequestParam` 变量。例如，`GreetingResponse` 中就使用的 `@RequestParam` 接受 name 变量
- `@RequestBody` 用来接收 body 中的变量，该变量一般是 JSON 对象。例如，本节的请求体 `UserRegisterRequest` 就是 json 对象。

- `@RequestHeader` 可以获取 Header 中的数据。一般不常用

除了 `@RequestBody` 只能出现一次外, `@RequestHeader` 和 `@RequestParam` 都可以出现多次。

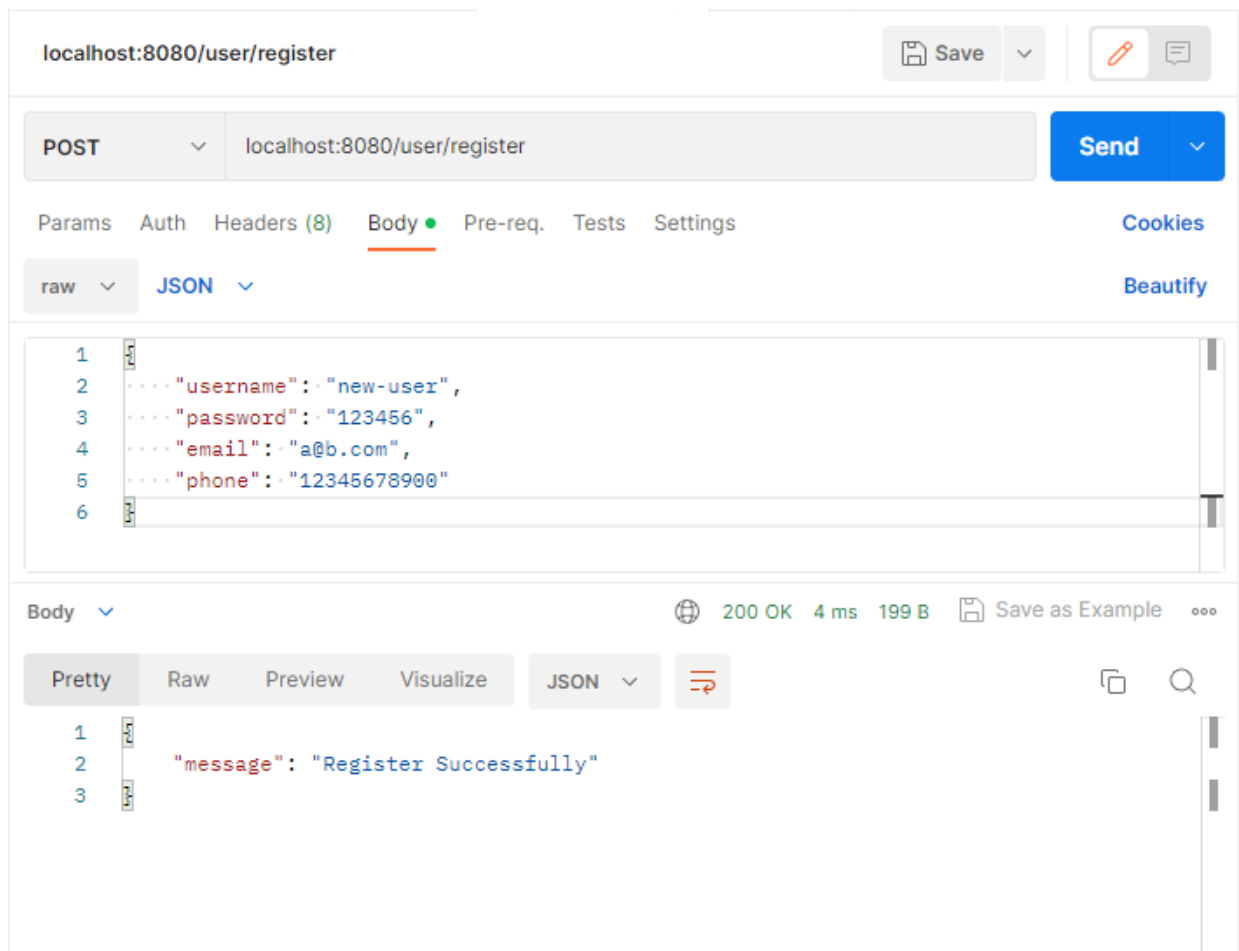
4. 测试接口

对于 GET 方法, 我们可以通过浏览器很方便地测试, 但是对于其他方法, 就无法这样测试了。因此推荐使用 Postman 进行接口测试。

Postman 可以从 Google 网上应用店中下载: <https://chrome.google.com/webstore/detail/postman/fhbjgibflinjbdggehcdcbncdddomop?hl=zh-CN>

Postman 使用方法简单说明, 以 `/user/register` 为例:

1. 首先指定 HTTP method 为 POST, 然后输入 url 为 localhost:8080/user/register。
2. 接下去填写 body 数据。选择 Body, 选择 raw, 在右侧类型选择 JSON(application/json), 然后输入 body 数据。
3. 最后点击 Send 发送请求, 如下图所示



用户身份认证机制

token 验证

Restful 服务的用户身份认证推荐使用 token 验证的方式。

Reference:

<https://ninghao.net/blog/2834>

<http://www.cnblogs.com/xiekeli/p/5607107.html>

简单来说，token 验证是在用户登录成功后将用户名、token 发行时间和过期时间等信息进行加密，并将生成的密文发回给客户端作为身份认证的凭证。这个 token 密文只有服务端能解密，当用户进行权限相关的操作时，需要将token 发回给服务端，服务端解密验证通过后进行相关操作。在这个过程中，服务端不需要储存任何用户登录状态，相比传统的 session 的方式，实现了服务端的无状态化。

session 验证

传统的 session 验证仍旧是一个优秀的身份认证机制。同学们可以使用 Spring Session，或者自己手动编写自定义的 session 管理代码来进行用户身份认证。

Spring Session: <http://projects.spring.io/spring-session/>

继续学习 Spring Boot

Get started: <https://spring.io/guides/gs/spring-boot/>

Full reference: <http://docs.spring.io/spring-boot/docs/2.0.0.BUILD-SNAPSHOT/reference/htmlsingle/#getting-started-installing-the-cli>

Learn more about spring: <https://spring.io/>

2. MyBatis

MyBatis 是一个后端数据库层面的框架，类似的框架还有 Hibernate。同学们可以在 PJ 中自由选择后端数据库层面的实现方式，但不太推荐

概述

MyBatis 是支持定制化 SQL、存储过程以及高级映射的优秀的持久层框架。MyBatis 避免了几乎所有的 JDBC 代码和手动设置参数以及获取结果集。MyBatis 可以对配置和原生Map使用简单的 XML 或注解，将接口和 Java 的POJOs(Plain Old Java Objects,普通的 Java对象)映射成数据库中的记录。

环境依赖

数据库

MyBatis 支持各种数据库，本次 Lab 中以 MySQL 为例。

没有安装MySQL的同学，先去[官网](#)下载安装

Maven

将下列两个依赖加入到 pom.xml 的 dependencies 中：

```
<dependency>
  <groupId>org.mybatis</groupId>
  <artifactId>mybatis</artifactId>
  <version>3.4.4</version>
</dependency>
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.25</version>
</dependency>
```

然后运行 `maven install` 安装这两个包。

官方文档

MyBatis 的官方文档将 MyBatis 的各方面阐述得很清楚，本 Lab 中不重复其中的内容。

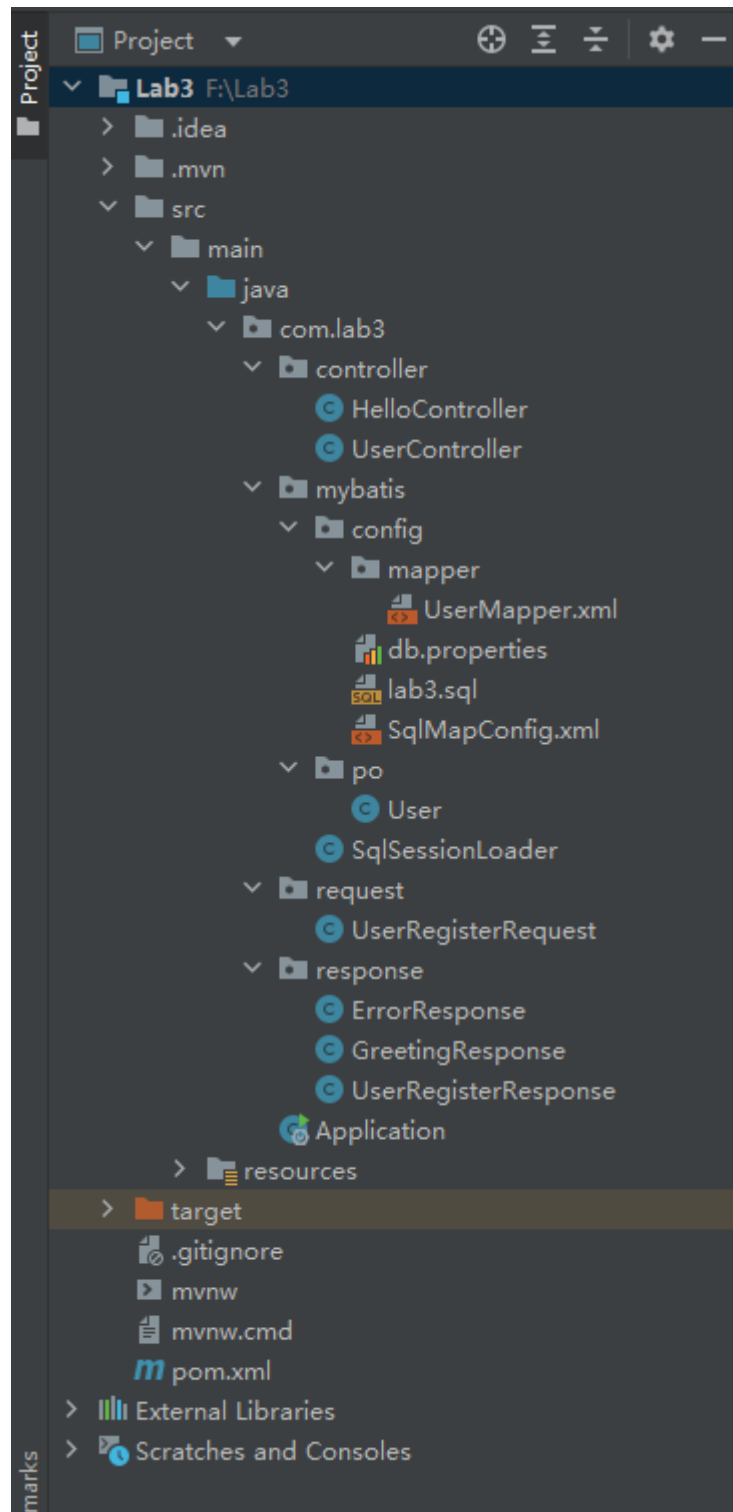
请同学们首先阅读 MyBatis 官方文档来学习 MyBatis。 Lab 将展示一个结合 Spring Boot 的样例程序。

中文: <http://www.mybatis.org/mybatis-3/zh/index.html>

英文: <http://www.mybatis.org/mybatis-3/index.html>

我们接着上文中 Spring Boot 创建的工程，将 MyBatis 加入进来。

最终的工程目录结构如图所示：



1.创建 MyBatis 核心配置文件 `mybatis/config/SqlMapConfig.xml` :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE configuration
    PUBLIC "-//mybatis.org//DTD Config 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-config.dtd">

<configuration>
    <properties resource="db.properties">

    </properties>
```

```

<environments default="development">
  <environment id="development">
    <transactionManager type="JDBC"/>
    <dataSource type="POOLED">
      <property name="driver" value="${driver}"/>
      <property name="url" value="${url}"/>
      <property name="username" value="${username}"/>
      <property name="password" value="${password}"/>
    </dataSource>
  </environment>
</environments>
<mappers>
  <mapper resource="mapper/UserMapper.xml"/>
</mappers>
</configuration>

```

2.properties 在 db.properties 文件中定义:

```

driver=com.mysql.jdbc.Driver
url=jdbc:mysql://localhost:3306/adweb_lab3?characterEncoding=utf-8
username=root
password=123456

```

3.对应地, 在 MySQL 中创建新的 Schema adweb-lab3。

4.使用 lab3.sql 创建表:

```

DROP TABLE IF EXISTS Article;
DROP TABLE IF EXISTS User;

CREATE TABLE IF NOT EXISTS User (
  userID INT(11) NOT NULL AUTO_INCREMENT,
  username VARCHAR(50) NOT NULL,
  password VARCHAR(50) NOT NULL,
  email VARCHAR(100) DEFAULT NULL,
  phone VARCHAR(20) DEFAULT NULL,
  PRIMARY KEY (userID)
);

INSERT INTO User (userID, username, password, email, phone) VALUES
  (1, 'kaiyudai', '12345678', 'kydai@fudan.edu.cn', '13666666666'),
  (2, 'fengshuangli', '12345678', '13302010002@fudan.edu.cn', '13888888888'),
  (3, 'zhongyitong', '12345678', NULL, NULL);

CREATE TABLE IF NOT EXISTS Article (
  articleID INT(11) NOT NULL AUTO_INCREMENT,
  userID INT(11) NOT NULL,
  title VARCHAR(100) NOT NULL,
  content VARCHAR(5000) NOT NULL,
  PRIMARY KEY (articleID),
  FOREIGN KEY (userID) REFERENCES User(userID)

```

```
);
```

5.新建 `mybatis/SqlSessionFactory.java` 来载入 MyBatis:

```
package com.lab3.mybatis;

import java.io.IOException;
import java.io.InputStream;

import org.apache.ibatis.io.Resources;
import org.apache.ibatis.session.SqlSession;
import org.apache.ibatis.session.SqlSessionFactory;
import org.apache.ibatis.session.SqlSessionFactoryBuilder;

public class SqlSessionFactory {
    private static SqlSessionFactory sqlSessionFactory;
    public static SqlSessionFactory getSqlSessionFactory() throws IOException {
        if (sqlSessionFactory == null) {
            InputStream inputStream =
                Resources.getResourceAsStream("SqlMapConfig.xml");
            sqlSessionFactory = new SqlSessionFactoryBuilder().build(inputStream);
        }
        return sqlSessionFactory;
    }
    public static SqlSession getSqlSession() throws IOException {
        return getSqlSessionFactory().openSession();
    }
}
```

6.为了使程序能够找到 `SqlMapConfig.xml` 文件, 在 `pom.xml` 的 `build` 标签中添加如下代码:

```
<resources>
    <resource>
        <directory>src/main/java/com/lab3/mybatis/config</directory>    //修改为自己工
程的路径
        <includes>
            <include>/**/*.xml</include>
            <include>/**/*.properties</include>
        </includes>
        <filtering>true</filtering>
    </resource>
</resources>
```

7.创建 `UserMapper.xml` 来定义对 User 表的操作映射:

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE mapper
    PUBLIC "-//mybatis.org//DTD Mapper 3.0//EN"
    "http://mybatis.org/dtd/mybatis-3-mapper.dtd">
<mapper namespace="com.lab3.UserMapper">    //修改为自己工程的路径
```

```

<select id="findUserById" parameterType="int"
        resultType="com.lab3.mybatis.po.User">
    select * from User where userID = #{userID}
</select>

<select id="findUserByUsername" parameterType="java.lang.String"
        resultType="com.lab3.mybatis.po.User">
    select * from User where username = #{username}
</select>

<insert id="addUser" parameterType="com.lab3.mybatis.po.User"
        useGeneratedKeys="true" keyProperty="userID">
    insert into User (username, password, email, phone)
    values (#{username}, #{password}, #{email}, #{phone})
</insert>

```

Mapper 是 MyBatis 核心功能，注意仔细阅读相关文档，理解 Mapper 的工作原理。

8.创建 Plain Object mybatis/po/User.java:

```

public class User {
    private int userID;
    private String username;
    private String password;
    private String email;
    private String phone;

    public User(int userID, String username, String password, String email, String
phone) {
        this.userID = userID;
        this.username = username;
        this.password = password;
        this.email = email;
        this.phone = phone;
    }

    public User(String username, String password, String email, String phone) {
        this.username = username;
        this.password = password;
        this.email = email;
        this.phone = phone;
    }

    public int getUserID() {
        return userID;
    }

    public void setUserID(int userID) {
        this.userID = userID;
    }

    /* 省略其他 setter 与 getter */
}

```

```
}
```

9.修改 `UserController.java` 中注册用户的对应代码:

```
@RestController
@CrossOrigin(origins = "*")
@RequestMapping("/user")
public class UserController {
    @RequestMapping(value = "/register", method = RequestMethod.POST)
    public @ResponseBody Object register(@RequestBody UserRegisterRequest request)
    throws IOException {
        SqlSession sqlSession = SqlSessionLoader.getSqlSession();

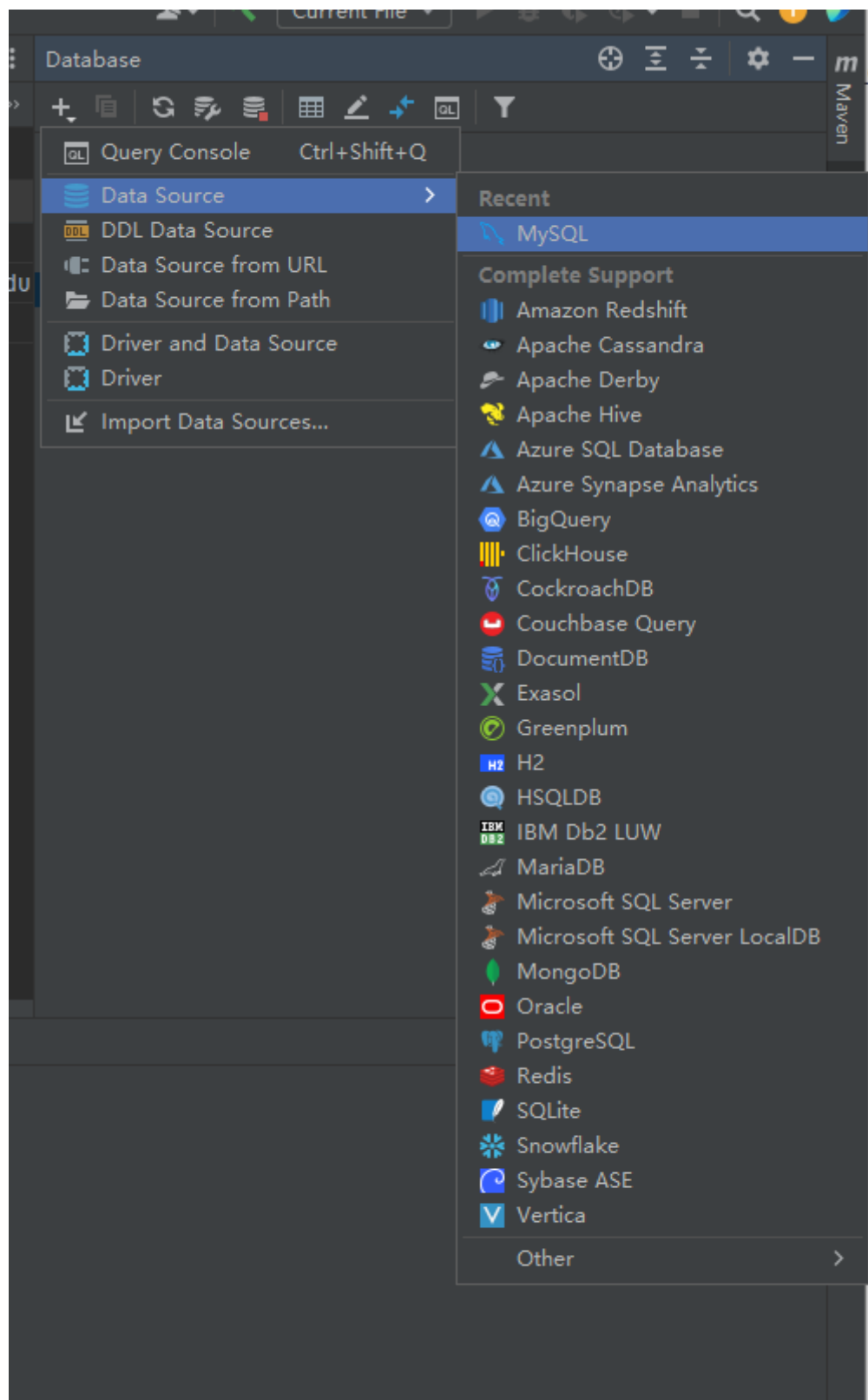
        User user = sqlSession.selectOne("com.lab3.UserMapper.findUserByUsername",
//修改为自己工程的路径
        request.getUsername());

        if (user != null) {
            sqlSession.close();
            return new ErrorResponse("The username is already used");
        } else {
            sqlSession.insert("com.lab3.UserMapper.addUser",          //修改为自己工程的
路径
            new User(request.getUsername(), request.getPassword(),
request.getEmail(), request.getPhone()));
            sqlSession.commit();
            sqlSession.close();
            return new UserRegisterResponse("Register Successfully!");
        }
    }
}
```

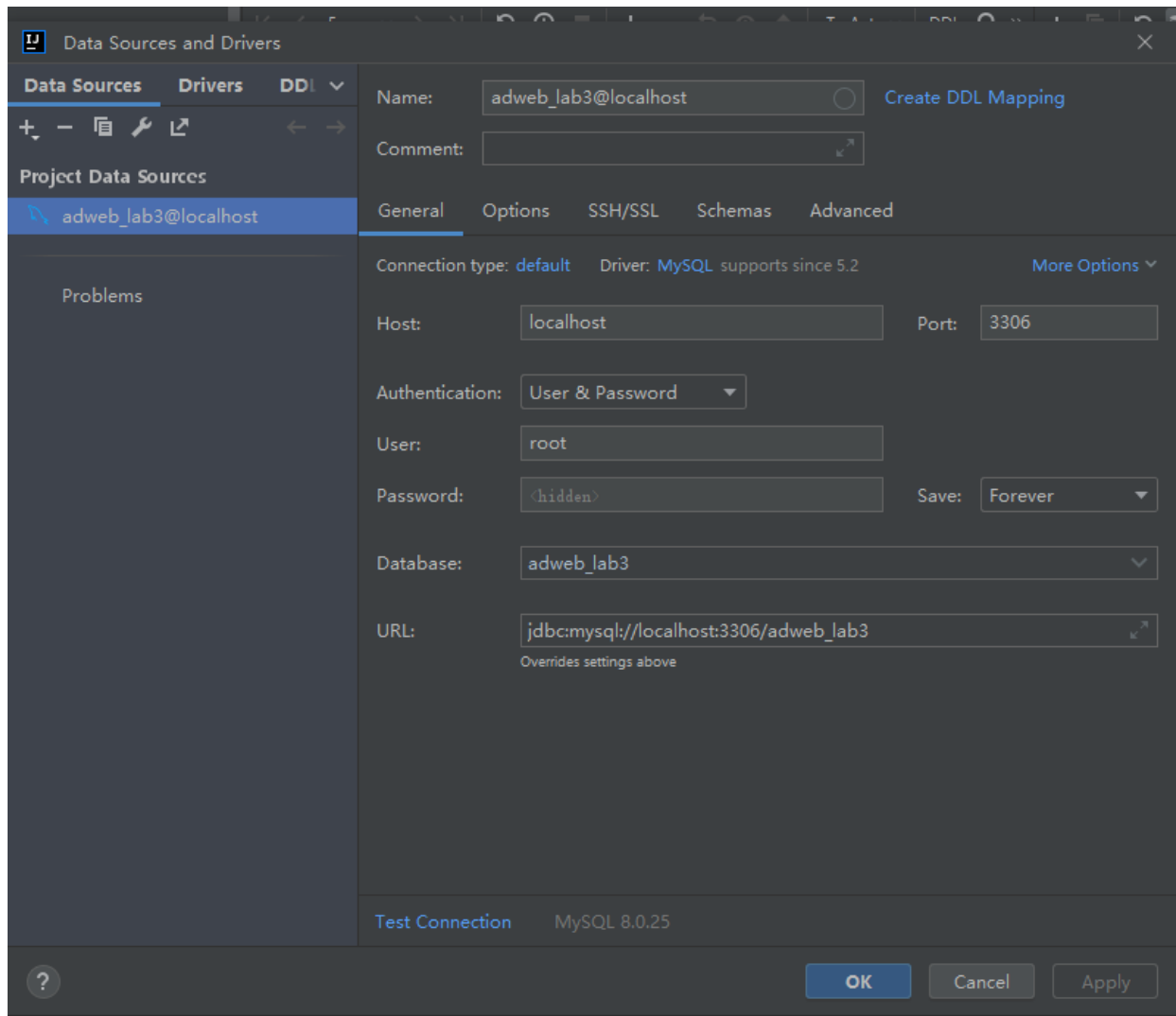
10. 测试数据库

首先将IDEA连接到MySQL, 具体步骤:

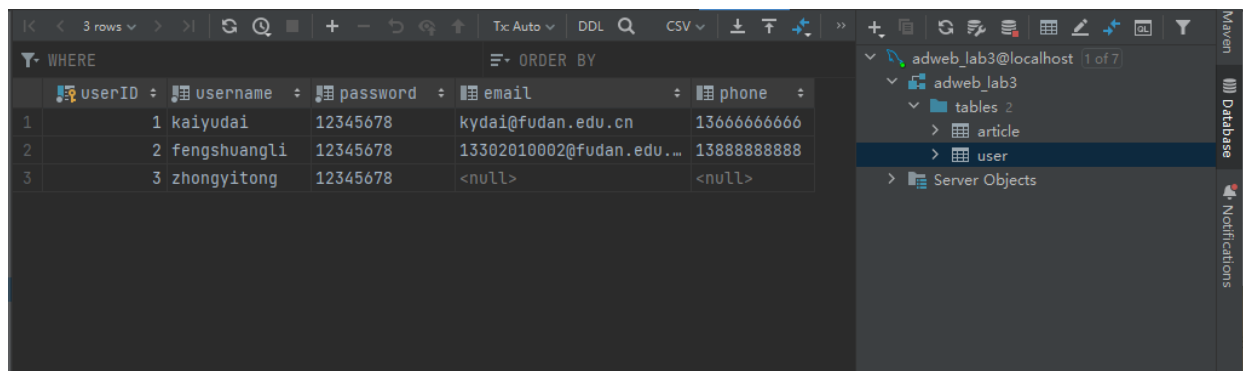
1) 点击右侧边栏Database, 选择mysql



2) 按照 `db.properties` 的设置连接mysql



3) 连接成功后, 可以访问 user 表



然后使用postman发送用户注册的post请求, 可以看到user表中新增一条数据:

localhost:8080/user/register

POST localhost:8080/user/register

Params Auth Headers (8) **Body** Pre-req. Tests Settings Cookies Beautify

raw JSON

```
1 {
2   "username": "new-user",
3   "password": "123456",
4   "email": "a@b.com",
5   "phone": "12345678900"
6 }
```

Body 200 OK 4 ms 199 B Save as Example

Pretty Raw Preview Visualize JSON

```
1 {
2   "message": "Register Successfully"
3 }
```

userID	username	password	email	phone
1	kaiyudai	12345678	kydai@fudan.edu.cn	13666666666
2	fengshuangli	12345678	13302010002@fudan.edu...	13888888888
3	zhongyitong	12345678	<null>	<null>
4	new-user	123456	a@b.com	12345678900

Database: adweb_lab3@localhost 1 of 7

- adweb_lab3
 - tables 2
 - article
 - user**
 - columns 5
 - keys 1
 - indexes 1
 - Server Objects

通过以上 10 个步骤，我们将 MyBatis 加入到了 Spring Boot 工程中，并进行了简单的运用。

以上代码有些地方写的比较粗糙，在结构上有改进空间。

3. 使用Angular HttpClient与后端通信

@angular/common/http 中的 HttpClient 类为 Angular 应用程序提供了一个简化的 API 来实现 HTTP 客户端功能。它基于浏览器提供的 XMLHttpRequest 接口。HttpClient 带来的其它优点包括：可测试性、强类型的请求和响应对象、发起请求与接收响应时的拦截器支持，以及更好的、基于可观察（Observable）对象的 API 以及流式错误处理机制。

1. app.module.ts 中导入HttpClientModule


```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [
    BrowserModule,
    HttpClientModule,    //新增行
    ...
  ],
  providers: ...
  bootstrap: [...]
})
```

2. 创建一个用户注册组件 `user-register`

```
ng generate component user-register
```

3. `user-register.component.ts` 引入HttpClient 并在构造函数声明

```
import { Component, OnInit } from '@angular/core';
import { HttpClient, HttpHeaders } from '@angular/common/http';

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

  constructor(public http:HttpClient) { }

  ngOnInit() {
  }

}
```

4. 定义 `greeting()`，调用 `get` 请求

```
greeting() {
  var api = 'http://localhost:8080/greeting';
  this.http.get(`${api}`).subscribe((response: any) => {
    window.alert(response.name);
  });
}
```

5. 定义 `register()`，调用 `post` 请求

```
export class UserRegisterComponent implements OnInit {
  private url = "http://localhost:8080/user/register";
  constructor(public http:HttpClient) { }
```

```

ngOnInit() {
}

register(){
  const httpOptions = {
    headers: new HttpHeaders({ 'Content-Type': 'application/json' })
  };
  this.http.post(this.url,
    {username:'abc',
      password: "123456",
      email: "a@b.com",
      phone: "12345678900"}, httpOptions).subscribe((response:any)=> {
    window.alert(response.message);
  });
}

greeting() {
  var api = 'http://localhost:8080/greeting';
  this.http.get(`${api}`).subscribe((response: any) => {
    window.alert(response.name);
  });
}
}

```

注意：调用api前，需检查是否存在跨域问题，即后端接口是否设置 `@CrossOrigin`

详情参考前文 *Spring Boot 支持跨域访问* 小节。

6. 在 `user-register.component.html` 中，定义 button 按钮绑定事件

```

<p>
  <button (click) = "greeting()">Hello world</button>
</p>
<p>
  <button (click)="register()">User Register</button>
</p>

```

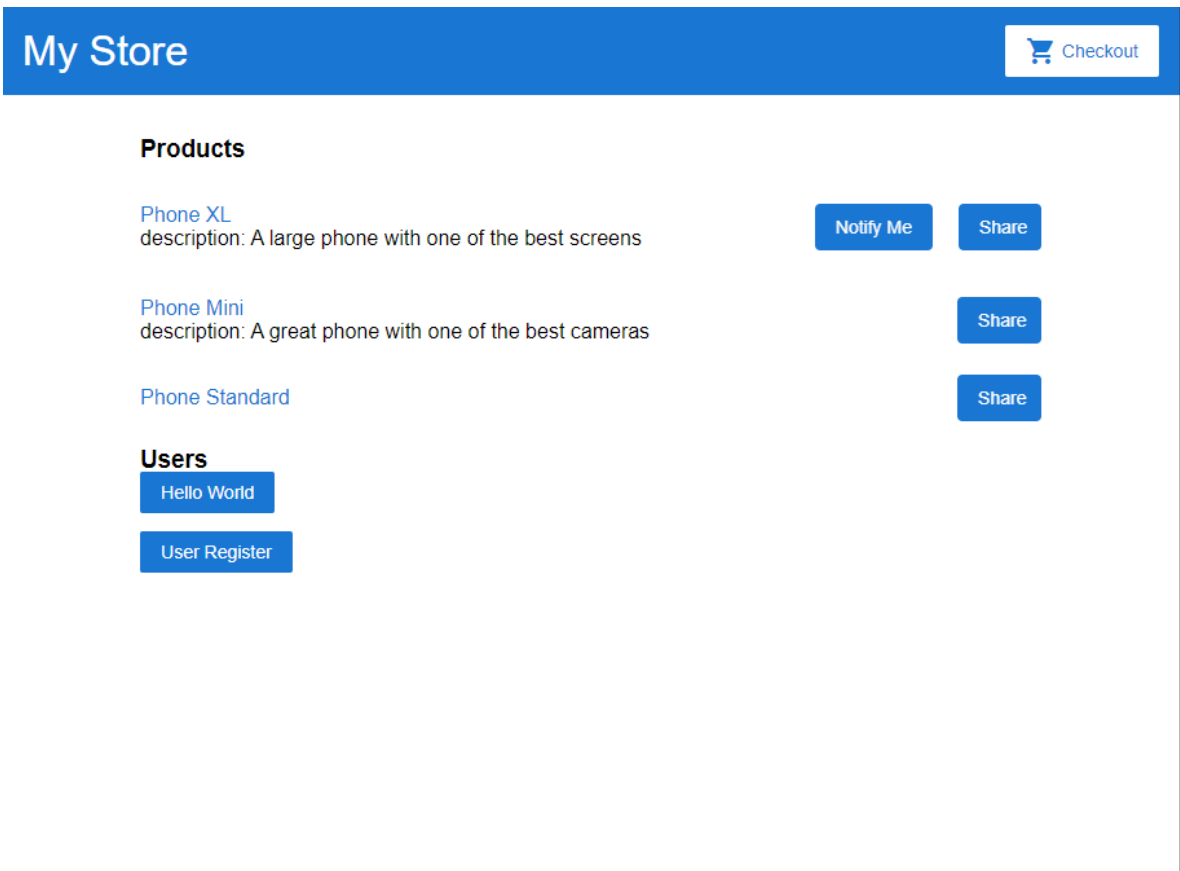
7. 在 `product-list.component.html` 上添加 `user-register` 组件

```

<h3 mat-subheader>Users</h3>
<mat-list-item>
  <app-user-register></app-user-register>
</mat-list-item>

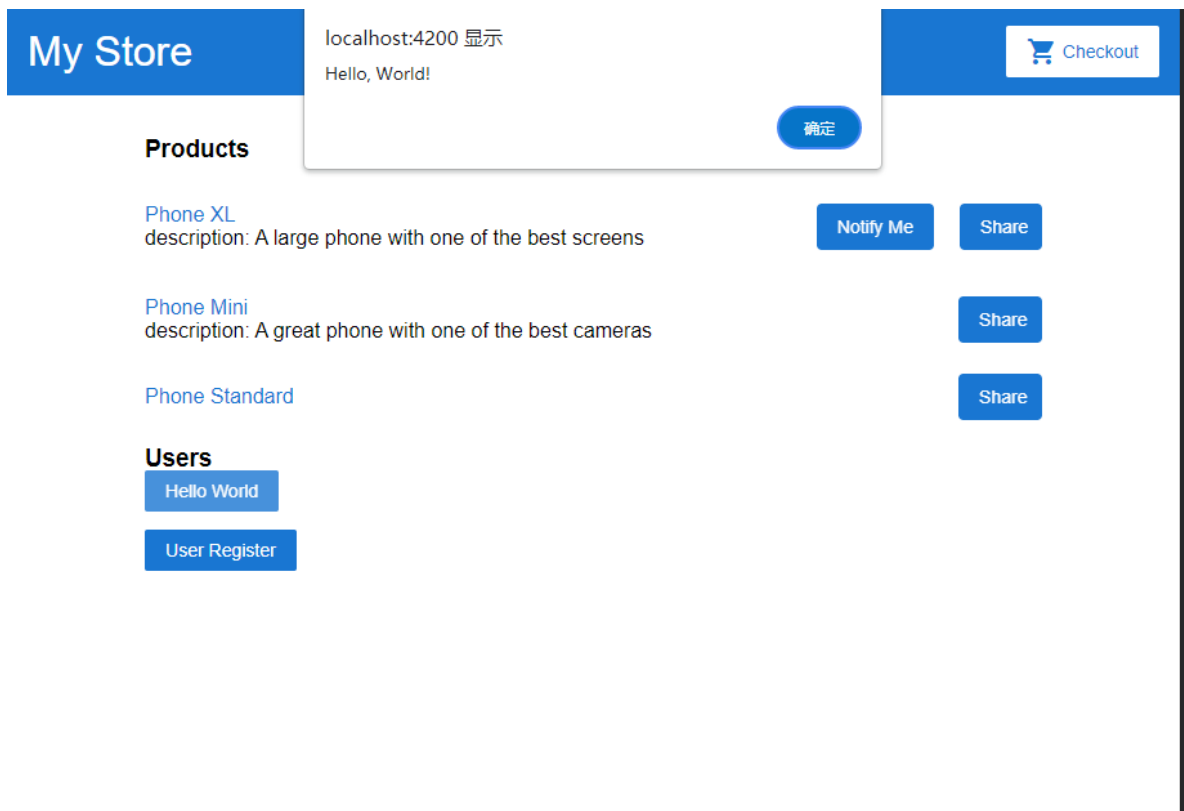
```

得到效果图：

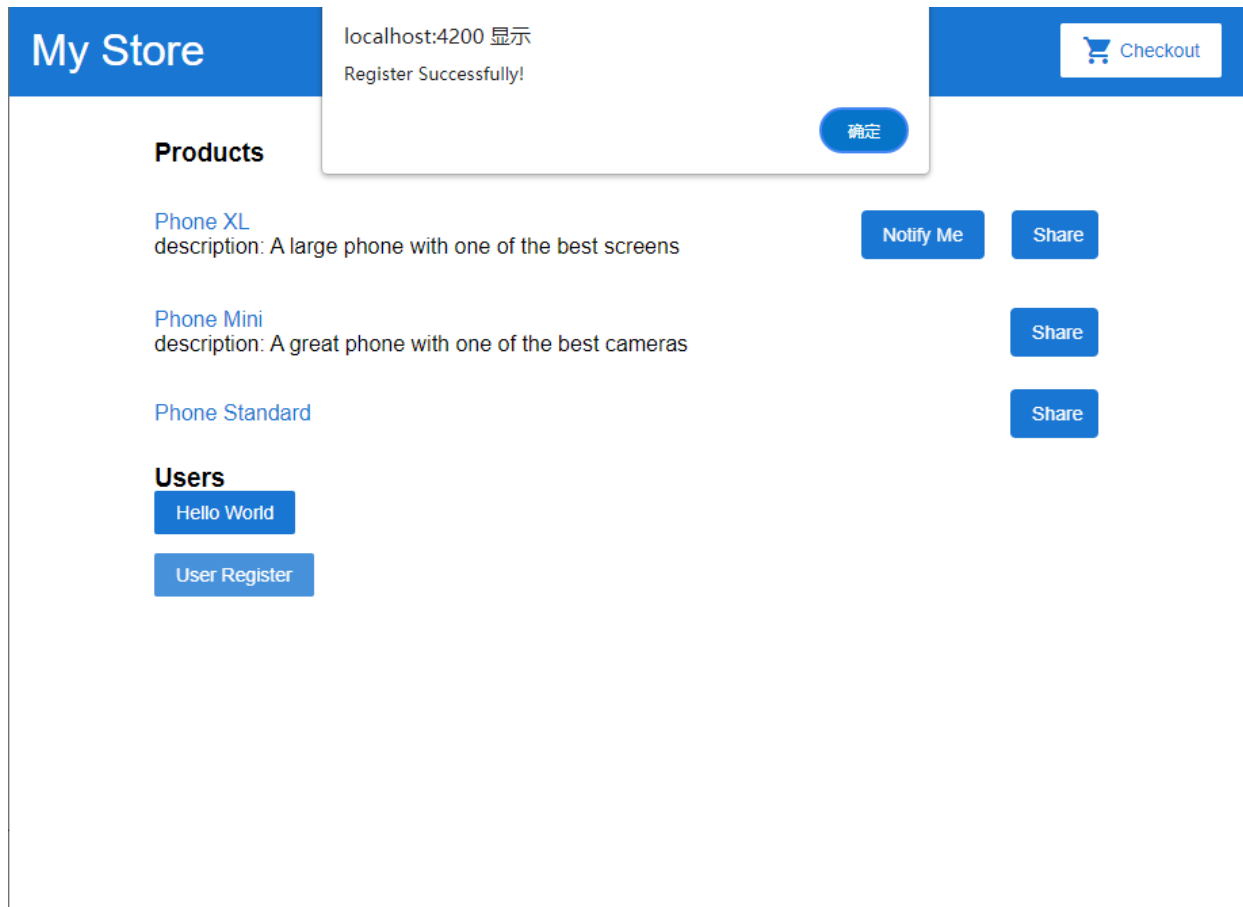


8. 测试接口

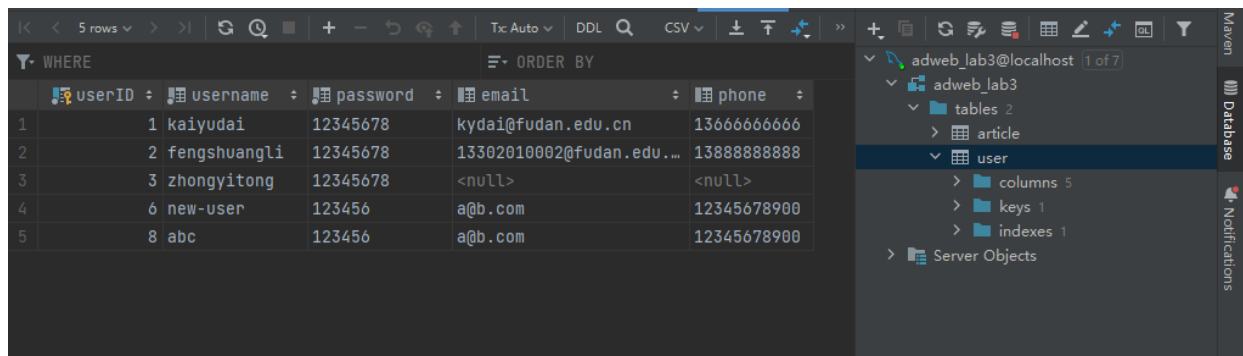
点击 `Hello world` 按钮，响应如图所示：



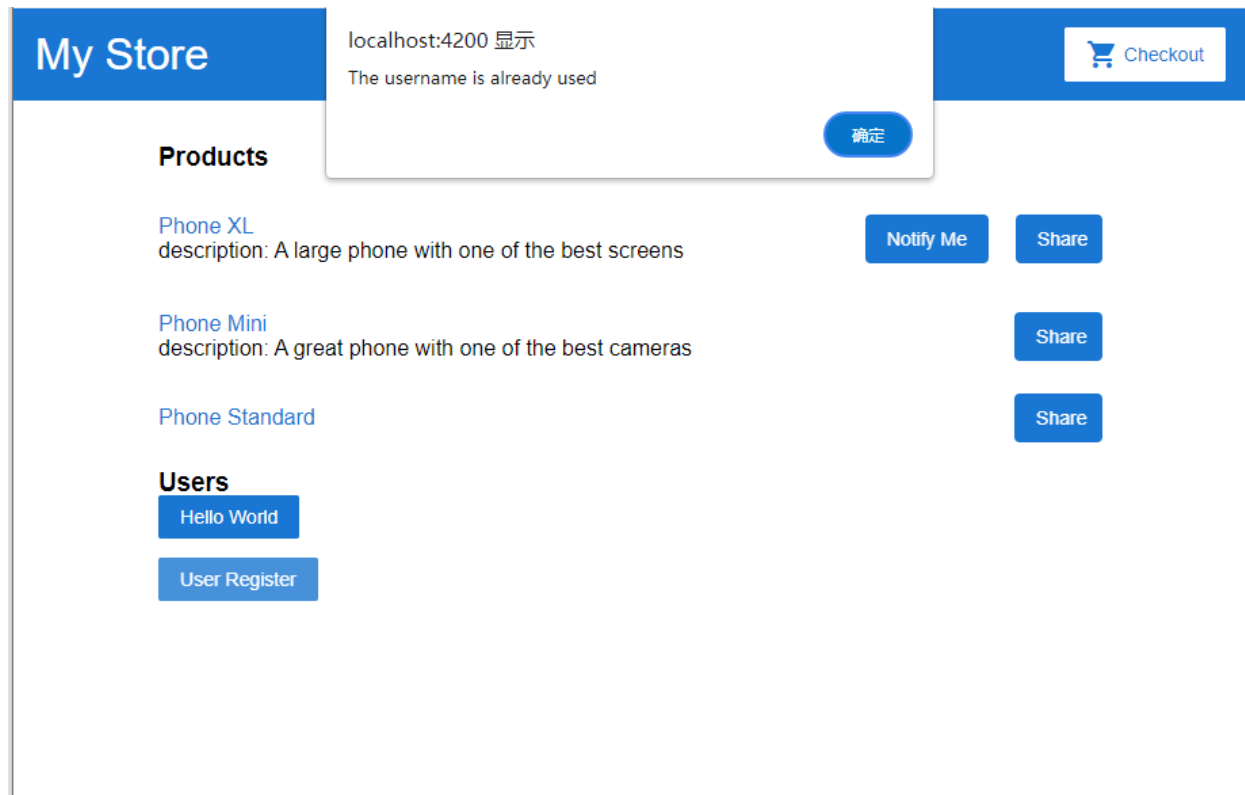
点击 `User Register` 按钮，响应如图所示：



并可以看到数据库 `user` 表中新增一行“abc”用户



此时，若再次点击 `User Register` 按钮，由于abc用户已注册成功，则会得到下图响应：



更多信息可参考 [Angular HTTP客户端](#)

4. 继续学习Spring Boot和MyBatis

目前，Lab工程已经完成了用户注册的流程，请同学们设计用户登录和列举所有用户这两个接口，结合 **SpringBoot** 和 **MyBatis** 编写对应的代码。

仍有剩余时间的同学可以尝试添加用户身份认证相关代码，继续学习 Spring Boot 和 MyBatis。