



# Spring Boot

IT管理方向

办公应用  
与协作

软件开发

数据库

运维方向

通用技能

云计算  
大数据

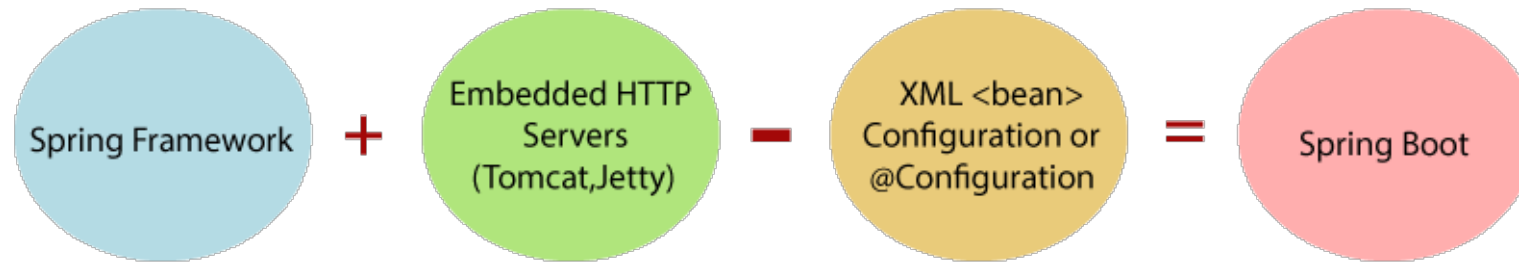
ERP工程师

云大物移智



关注微信公众号  
享终身免费学习

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. This chapter will give you an introduction to Spring Boot and familiarizes you with its basic concepts.

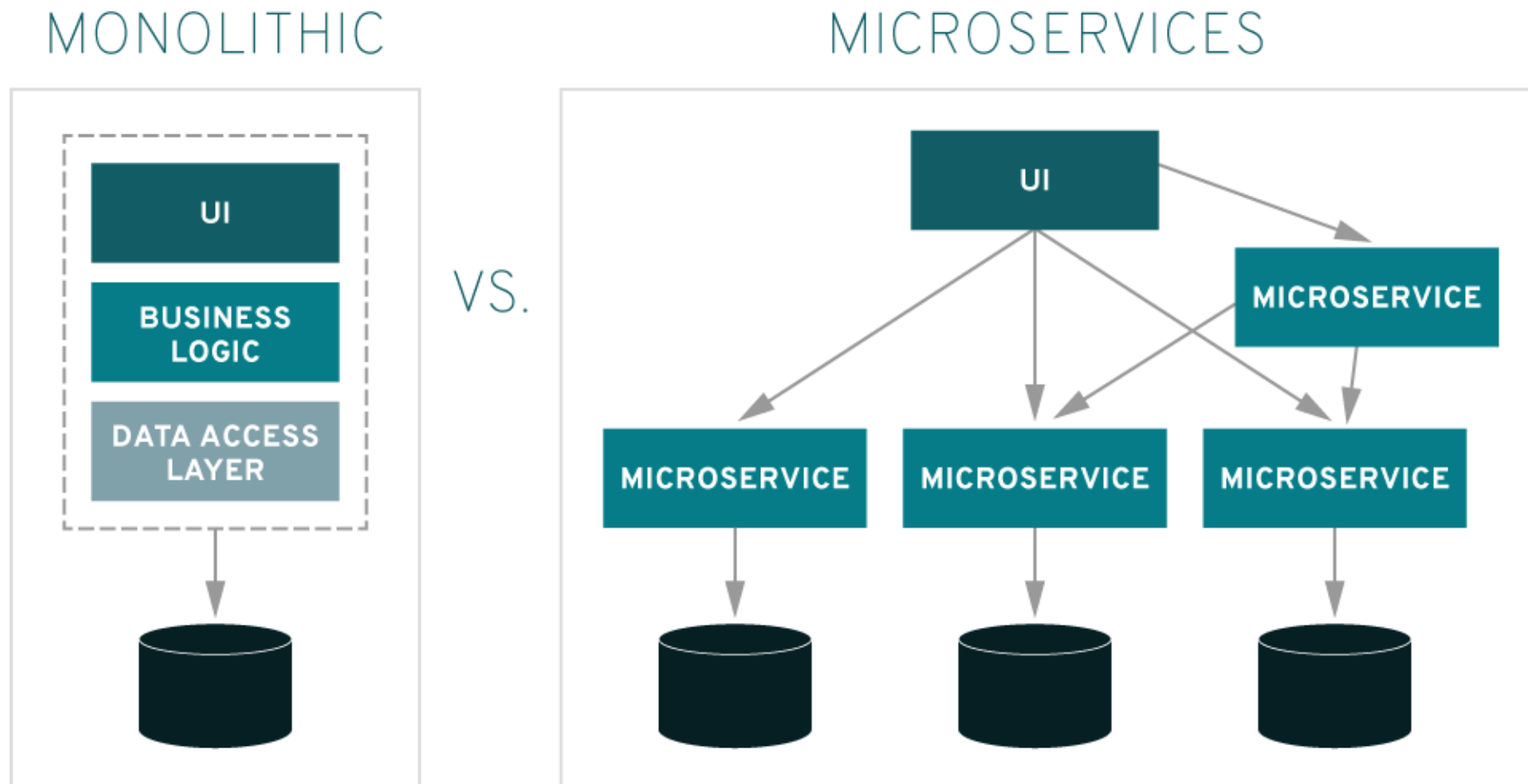


In short, Spring Boot is the combination of **Spring Framework** and **Embedded Servers**.

In Spring Boot, there is no requirement for XML configuration (deployment descriptor). It uses convention over configuration software design paradigm that means it decreases the effort of the developer.

## 1. What is Micro Service?

Micro Service is an architecture that allows the developers to develop and deploy services independently. Each service running has its own process and this achieves the lightweight model to support business applications.



图一、 Monolithic architecture vs microservices architecture

## 1.1. Advantages

Micro services offers the following advantages to its developers –

- Easy deployment
- Simple scalability
- Compatible with Containers
- Minimum configuration
- Lesser production time

---

## 2. What is Spring Boot?

Spring Boot provides a good platform for Java developers to develop a stand-alone and production-grade spring application that you can **just run**. You can get started with minimum configurations without the need for an entire Spring configuration setup.

## 2.1. Advantages

Spring Boot offers the following advantages to its developers –

- Easy to understand and develop spring applications
- Increases productivity
- Reduces the development time

## 2.2. Goals

Spring Boot is designed with the following goals –

- To avoid complex XML configuration in Spring
- To develop a production ready Spring applications in an easier way
- To reduce the development time and run the application independently
- Offer an easier way of getting started with the application

## 3. Why Spring Boot?

You can choose Spring Boot because of the features and benefits it offers as given here –

- It provides a flexible way to configure Java Beans, XML configurations, and Database Transactions.
- It provides a powerful batch processing and manages REST endpoints.
- In Spring Boot, everything is auto configured; no manual configurations are needed.
- It offers annotation-based spring application
- Eases dependency management
- It includes Embedded Servlet Container



### 4. How does it work?

Spring Boot automatically configures your application based on the dependencies you have added to the project by using `@EnableAutoConfiguration` annotation. For example, if MySQL database is on your classpath, but you have not configured any database connection, then Spring Boot auto-configures an in-memory database.

The entry point of the spring boot application is the class contains `@SpringBootApplication` annotation and the main method.

Spring Boot automatically scans all the components included in the project by using `@ComponentScan` annotation.

## 5. Spring Boot Starters

Handling dependency management is a difficult task for big projects. Spring Boot resolves this problem by providing a set of dependencies for developers convenience.

For example, if you want to use Spring and JPA for database access, it is sufficient if you include **spring-boot-starter-data-jpa** dependency in your project.

Note that all Spring Boot starters follow the same naming pattern **spring-boot-starter-**, *where* indicates that it is a type of the application.

Look at the following Spring Boot starters explained below for a better understanding –

**Spring Boot Starter Actuator dependency** is used to monitor and manage your application. Its code is shown below –

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

**Spring Boot Starter Security dependency** is used for Spring Security. Its code is shown below –

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

**Spring Boot Starter web dependency** is used to write a Rest Endpoints. Its code is shown below –

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

**Spring Boot Starter Thyme Leaf dependency** is used to create a web application. Its code is shown below –

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

**Spring Boot Starter Test dependency** is used for writing Test cases. Its code is shown below –

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-test</artifactId>
</dependency>
```

## 6. Auto Configuration

Spring Boot Auto Configuration automatically configures your Spring application based on the JAR dependencies you added in the project. For example, if MySQL database is on your class path, but you have not configured any database connection, then Spring Boot auto configures an in-memory database.

For this purpose, you need to add `@EnableAutoConfiguration` annotation or `@SpringBootApplication` annotation to your main class file. Then, your Spring Boot application will be automatically configured.

Observe the following code for a better understanding –

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

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

## 7. Spring Boot Application

The entry point of the Spring Boot Application is the class contains `@SpringBootApplication` annotation. This class should have the main method to run the Spring Boot application. `@SpringBootApplication` annotation includes Auto- Configuration, Component Scan, and Spring Boot Configuration.

If you added `@SpringBootApplication` annotation to the class, you do not need to add the `@EnableAutoConfiguration`, `@ComponentScan` and `@SpringBootConfiguration` annotation. The `@SpringBootApplication` annotation includes all other annotations.

Observe the following code for a better understanding –

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

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

## 8. Component Scan

Spring Boot application scans all the beans and package declarations when the application initializes. You need to add the [@ComponentScan](#) annotation for your class file to scan your components added in your project.

Observe the following code for a better understanding –

```
import org.springframework.boot.SpringApplication;  
import org.springframework.context.annotation.ComponentScan;  
  
@ComponentScan  
public class DemoApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(DemoApplication.class, args);  
    }  
}
```

---

## 9. Quick Start

## 9.1. Maven

### pom.xml

```
<?xml version = "1.0" encoding = "UTF-8"?>
<project xmlns = "http://maven.apache.org/POM/4.0.0"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>com.tutorialspoint</groupId>
  <artifactId>demo</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>jar</packaging>

  <name>demo</name>
  <description>Demo project for Spring Boot</description>

  <parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>1.5.8.RELEASE</version>
    <relativePath/> <!-- lookup parent from repository -->
  </parent>
```



```
<properties>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <java.version>1.8</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-maven-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

```
</project>
```

Spring Boot provides a number of **Starters** to add the jars in our class path. For example, for writing a Rest Endpoint, we need to add the **spring-boot-starter-web** dependency in our class path.

## 9.2. Main Method

The main method should be writing the Spring Boot Application class. This class should be annotated with **@SpringBootApplication**. This is the entry point of the spring boot application to start. You can find the main class file under **src/java/main** directories with the default package.

In this example, the main class file is located at the **src/java/main** directories with the default package **com.tutorialspoint.demo**. Observe the code shown here for a better understanding –

```
package com.test.demo;

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

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

## 9.3. Write a Rest Endpoint

To write a simple Hello World Rest Endpoint in the Spring Boot Application main class file itself, follow the steps shown below –

- Firstly, add the **@RestController** annotation at the top of the class.
- Now, write a Request URI method with **@RequestMapping** annotation.
- Then, the Request URI method should return the **Hello World** string.

Now, your main Spring Boot Application class file will look like as shown in the code given below –

```
package com.test.demo;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

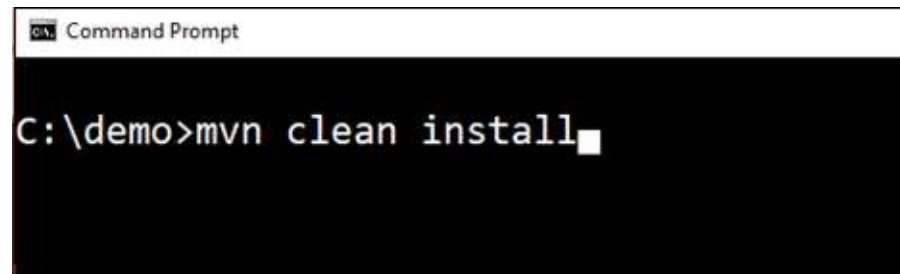
@SpringBootApplication
@RestController

public class DemoApplication {
    public static void main(String[] args) {
        SpringApplication.run(DemoApplication.class, args);
    }
    @RequestMapping(value = "/")
    public String hello() {
        return "Hello World";
    }
}
```

## 9.4. Create an Executable JAR

Let us create an executable JAR file to run the Spring Boot application by using Maven and Gradle commands in the command prompt as shown below –

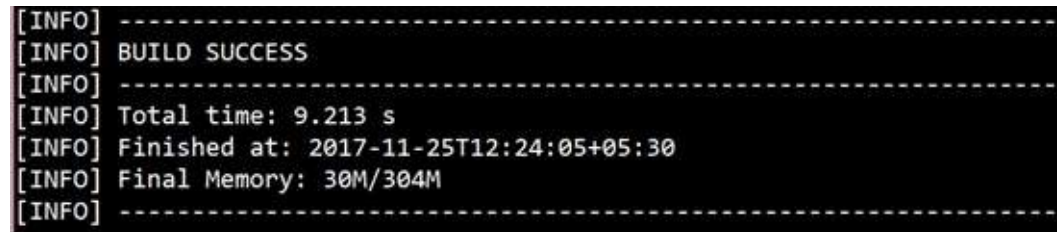
Use the Maven command `mvn clean install` as shown below –



```
Command Prompt
C:\demo>mvn clean install
```

图二、Command MVN Clean Install

After executing the command, you can see the **BUILD SUCCESS** message at the command prompt as shown below –

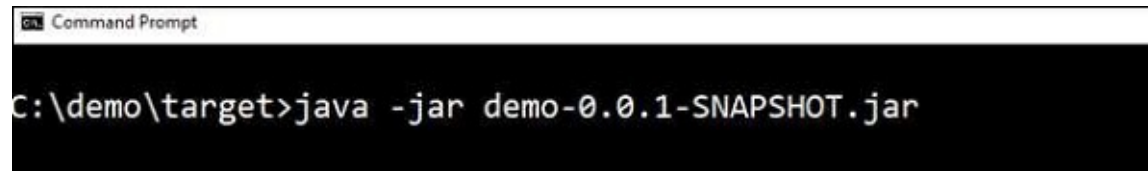


```
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 9.213 s
[INFO] Finished at: 2017-11-25T12:24:05+05:30
[INFO] Final Memory: 30M/304M
[INFO] -----
```

图三、BUILD SUCCESS Message

## 9.5. Run Hello World with Java

Now, run the JAR file by using the command **java -jar** . Observe that in the above example, the JAR file is named **demo-0.0.1-SNAPSHOT.jar**



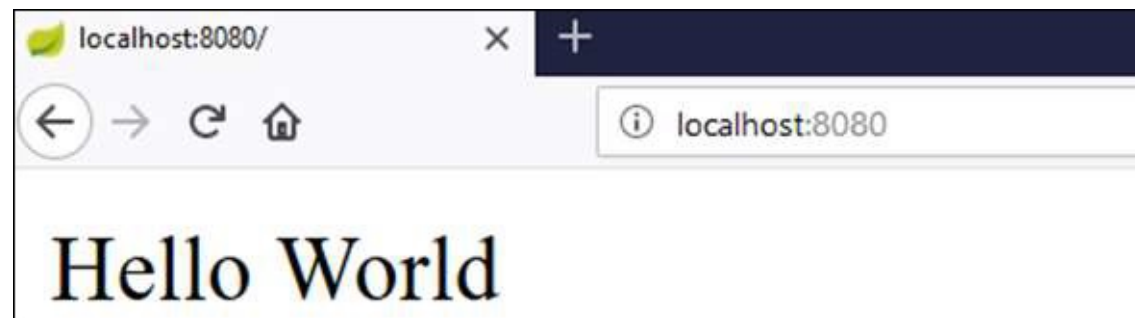
图四、JAR File Named Demo SNAPSHOT

Once you run the jar file, you can see the output in the console window as shown below –



图五、Output in Console Window

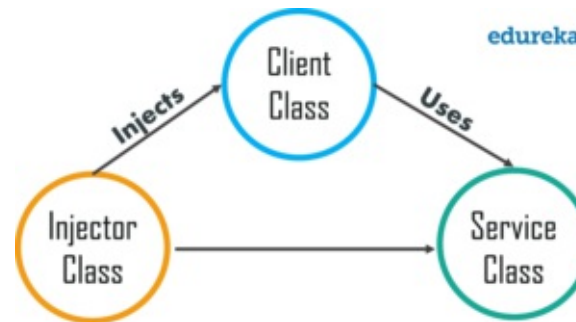
Now, look at the console, Tomcat started on port 8080 (http). Now, go to the web browser and hit the URL **http://localhost:8080/** and you can see the output as shown below –



图六、Tomcat Started on Port 8080 (http).

## 10. Dependency Injection

Dependency Injection is the ability of an object to supply dependencies of another object.



- **Client Class:** This is the dependent class and is dependent on the Service class.
- **Service Class:** This class provides a service to the client class.
- **Injector Class:** This class is responsible for injecting the service class object into the client class



## 10.1. The World Before Dependency Injection

Let's take a look at a counter example

*// DON'T DO THIS*

```
public class BadExample {

    public static void main(String[] args) {
        new ComputerProcessor()
            .addComputer(new Doubler())
            .addComputer(new Squarer())
            .computeAll(8);
    }

    static class ComputerProcessor {

        private List computers = new ArrayList();

        public ComputerProcessor addComputer(Object o) {
            computers.add(o);
            return ComputerProcessor.this;
        }

        public void computeAll(long value) {
            for (Object o : computers) {
                long computedValue = -1;
```

```
        if (o instanceof Doubler) {
            computedValue = ((Doubler) o).computeDouble(value);
        } else if (o instanceof Squarer) {
            computedValue = ((Squarer) o).computeSquare(value);
        }
        String name = o.getClass().getSimpleName();
        System.out.println("Computer: " + name + ", value: " + value + " computed value: " + computedValue);
    }
}

static class Doubler {

    public long computeDouble(long value) {
        return value*2;
    }
}

static class Squarer {

    public long computeSquare(long value) {
        return value*value;
    }
}
}
```

The `ComputerProcessor` is very fragile in this example. If we add a new class called `Cuber`, and we want to be able to represent it in `ComputerProcessor`, we need another if statement and more casting in order to call its `computeCube` method.

Here's the same example using Inversion of Control:

```
public class IoCExample {

    public static void main(String[] args) {
        new ComputerProcessor()
            .addComputer(new Doubler())
            .addComputer(new Squarer())
            .computeAll(8);
    }

    static class ComputerProcessor {

        private List<Computer> computers = new ArrayList<>();

        public ComputerProcessor addComputer(Computer c) {
            computers.add(c);
            return ComputerProcessor.this;
        }

        public void computeAll(long value) {
            for (Computer c : computers) {
                String name = c.getClass().getSimpleName();
                System.out.println("Computer: " + name + ", value: " + value + " computed value: " + c.compute(value));
            }
        }
    }
}
```

```
    }  
}  
  
interface Computer {  
  
    long compute(long value);  
}  
  
static class Doubler implements Computer {  
  
    public long compute(long value) {  
        return value*2;  
    }  
}  
  
static class Squarer implements Computer {  
  
    public long compute(long value) {  
        return value*value;  
    }  
}  
}
```

The enabling technology here is the `Computer` interface. Now, `ComputerProcessor` only deals with objects that conform to the `Computer` interface. If we want to add a new `Cuber` class, all it has to do is implement the `Computer` interface. Nothing need change in `ComputerProcessor` to deal with `Cuber` objects.

---

Dependency Injection takes this concept and systematizes it in a way that makes it even easier to interact with your interfaces. Throughout this post, I will be using a Spring Boot example to demonstrate Dependency Injection.

## 10.2. Dependency Injection, Spring Style

Let's jump into a simple example and then break it down:

```
@RestController
public class HomeController {

    @Autowired
    GreetingService greetingService;

    @RequestMapping("/")
    public String home() {
        return greetingService.greet();
    }
}
```

Spring introduced the `@Autowired` annotation for dependency injection. Any of the Spring components can be autowired. These include, components, configurations, services and beans. We'll look at a few of these in detail below.

It's a common pattern for controllers to be responsible for managing requests and responses while services perform business logic. Let's look at our `GreetingService`:

```
public interface GreetingService {

    String greet();
}
```

Here's an implementation class:

```
@Service
public class EnglishGreetingService implements GreetingService {

    @Override
    public String greet() {
        return "Hello World!";
    }
}
```

The `@Service` annotation makes it autowireable. Spring injects the dependency into our controller. If I set a breakpoint, I can see the implementation class backing `greetingService`:

Together, this setup adheres to the Dependency Inversion Principle. If the internals of the implementation class change, we don't need to touch `HomeController`.

Now, let's say I want to support another `GreetingService` implementation. Say, `FrenchGreetingService`:

```
@Service
public class FrenchGreetingService implements GreetingService {

    @Override
    public String greet() {
        return "Bonjour Monde!";
    }
}
```

If I try to fire up my Spring Boot app now, I will get an error:

Caused by: org.springframework.beans.factory.NoUniqueBeanDefinitionException: No qualifying bean of type [com.stormpath.example.service.GreetingService] is defined: expected single matching bean but found 2: englishGreetingService,frenchGreetingService

Spring Boot has no way of knowing which one of my implementation classes I want to inject into the `HomeController`. However, it does have a number of powerful annotations to deal with this. Let's say that I want to inject the language specific version of the greeting service based on a property setting. We can use a `Configuration` to tell Spring Boot which Service we want.

## @Configuration

```
public class GreetingServiceConfig {
```

### @Bean

```
@ConditionalOnProperty(name = "language.name", havingValue = "english", matchIfMissing = true)
```

```
public GreetingService englishGreetingService() {
```

```
    return new EnglishGreetingService();
```

```
}
```

### @Bean

```
@ConditionalOnProperty(name = "language.name", havingValue = "french")
```

```
public GreetingService frenchGreetingService() {
```

```
    return new FrenchGreetingService();
```

```
}
```

```
}
```



```
mvn clean install
```

*# English*

```
java -jar target/*.jar &
```

```
http localhost:8080
```

*# Also English*

```
LANGUAGE_NAME=english java -jar target/*.jar &
```

```
http localhost:8080
```

*# French*

```
LANGUAGE_NAME=french java -jar target/*.jar &
```

```
http localhost:8080
```

## 11. Building RESTful Web Services

### The Spring Boot main application class – DemoApplication.java

```
package com.tutorialspoint.demo;

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

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

### The POJO class – Product.java

```
package com.tutorialspoint.demo.model;
```

```
public class Product {  
    private String id;  
    private String name;  
  
    public String getId() {  
        return id;  
    }  
    public void setId(String id) {  
        this.id = id;  
    }  
    public String getName() {  
        return name;  
    }  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

## The Rest Controller class – ProductServiceController.java

```
package com.tutorialspoint.demo.controller;
```

```
import java.util.HashMap;
```

```
import java.util.Map;
```

```
import org.springframework.http.HttpStatus;
```

```
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RestController;

import com.tutorialspoint.demo.model.Product;

@RestController

public class ProductServiceController {

    private static Map<String, Product> productRepo = new HashMap<>();
    static {
        Product honey = new Product();
        honey.setId("1");
        honey.setName("Honey");
        productRepo.put(honey.getId(), honey);

        Product almond = new Product();
        almond.setId("2");
        almond.setName("Almond");
        productRepo.put(almond.getId(), almond);
    }

    @RequestMapping(value = "/products/{id}", method = RequestMethod.DELETE)
    public ResponseEntity<Object> delete(@PathVariable("id") String id) {
        productRepo.remove(id);
    }
}
```

```
productRepo.remove(id);  
return new ResponseEntity<>("Product is deleted successssfully", HttpStatus.OK);  
}  
  
@RequestMapping(value = "/products/{id}", method = RequestMethod.PUT)  
public ResponseEntity<Object> updateProduct(@PathVariable("id") String id, @RequestBody Product product) {  
    productRepo.remove(id);  
    product.setId(id);  
    productRepo.put(id, product);  
    return new ResponseEntity<>("Product is updated successssfully", HttpStatus.OK);  
}  
  
@RequestMapping(value = "/products", method = RequestMethod.POST)  
public ResponseEntity<Object> createProduct(@RequestBody Product product) {  
    productRepo.put(product.getId(), product);  
    return new ResponseEntity<>("Product is created successfully", HttpStatus.CREATED);  
}  
  
@RequestMapping(value = "/products")  
public ResponseEntity<Object> getProduct() {  
    return new ResponseEntity<>(productRepo.values(), HttpStatus.OK);  
}  
  
// using spring JDBC template  
@RequestMapping("/jdbc-products")  
@ResponseBody  
public List<Map<String, Object>> products() {  
    String sql = "select * from products";  
    List<Map<String, Object>> products = new ArrayList<>();  
    try {  
        ResultSet rs = jdbcTemplate.query(sql, new BeanPropertyRowMapper<Product>(Product.class));  
        for (Product product : rs) {  
            Map<String, Object> productMap = new HashMap<>();  
            productMap.put("id", product.getId());  
            productMap.put("name", product.getName());  
            productMap.put("price", product.getPrice());  
            products.add(productMap);  
        }  
    } catch (SQLException e) {  
        e.printStackTrace();  
    }  
    return products;  
}
```

```
String sql = "select * from product";  
List<Map<String, Object>> list = jdbcTemplate.queryForList(sql);  
return list;  
}  
}
```

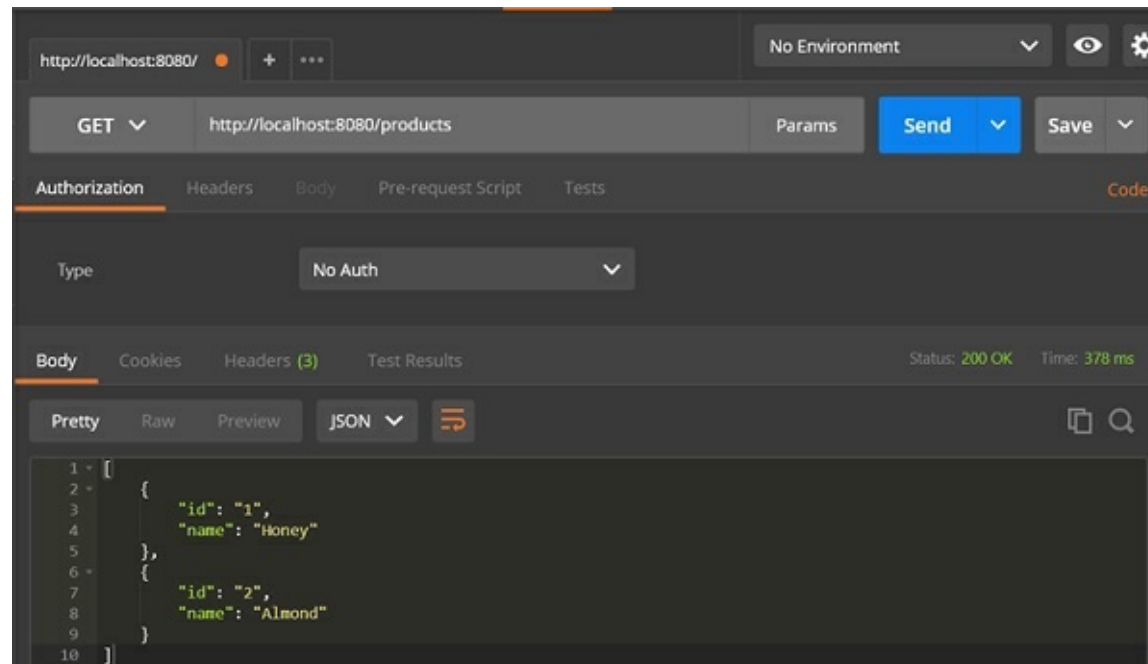
You can create an executable JAR file, and run the spring boot application by using the below Maven or Gradle commands as shown –

For Maven, use the command shown below –

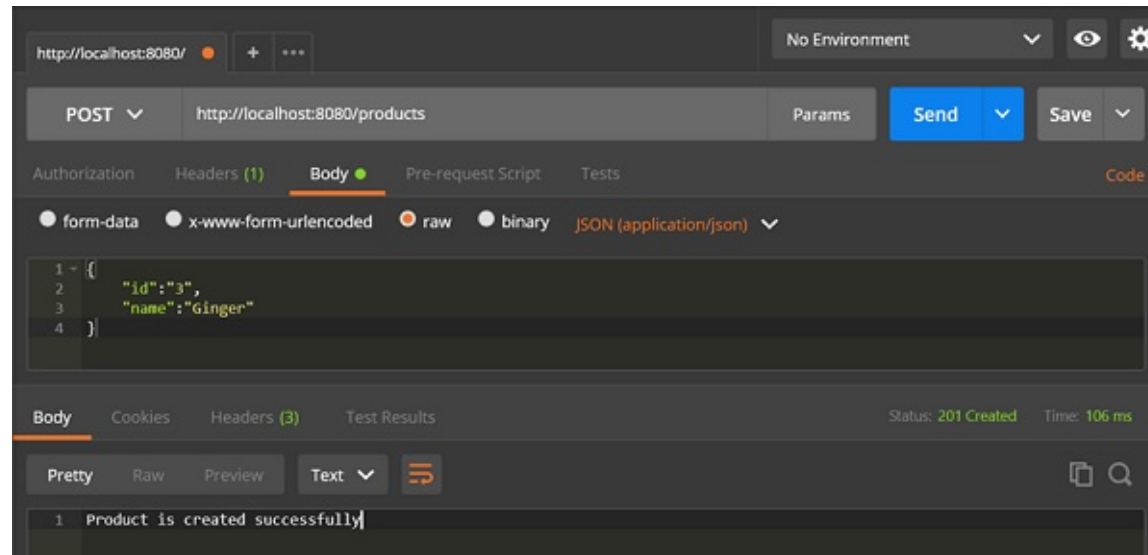
```
mvn clean install
```

Now hit the URL shown below in POSTMAN application and see the output.

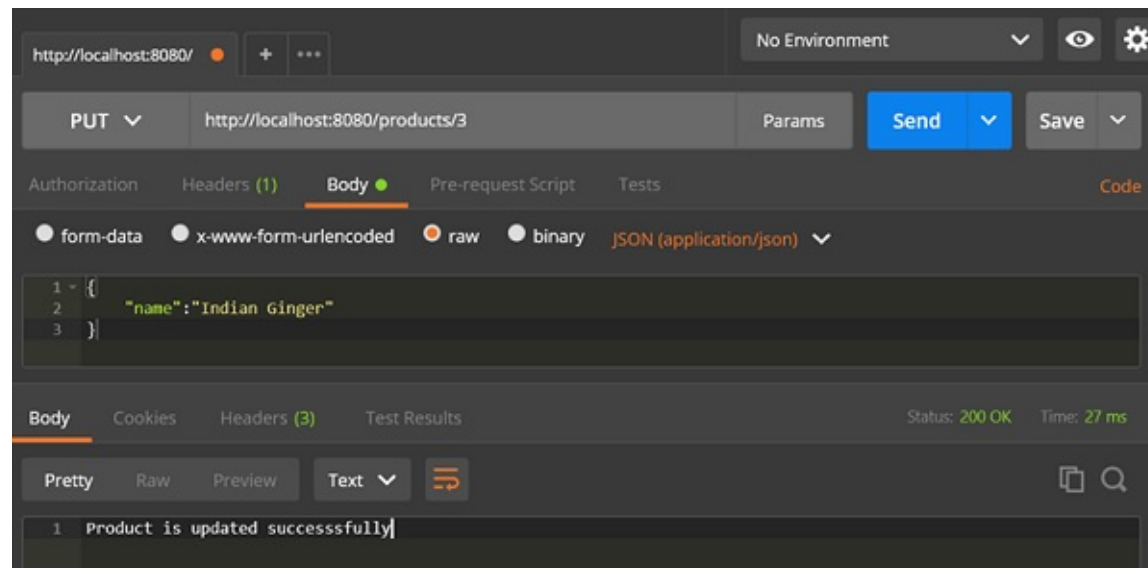
GET API URL is: <http://localhost:8080/products>



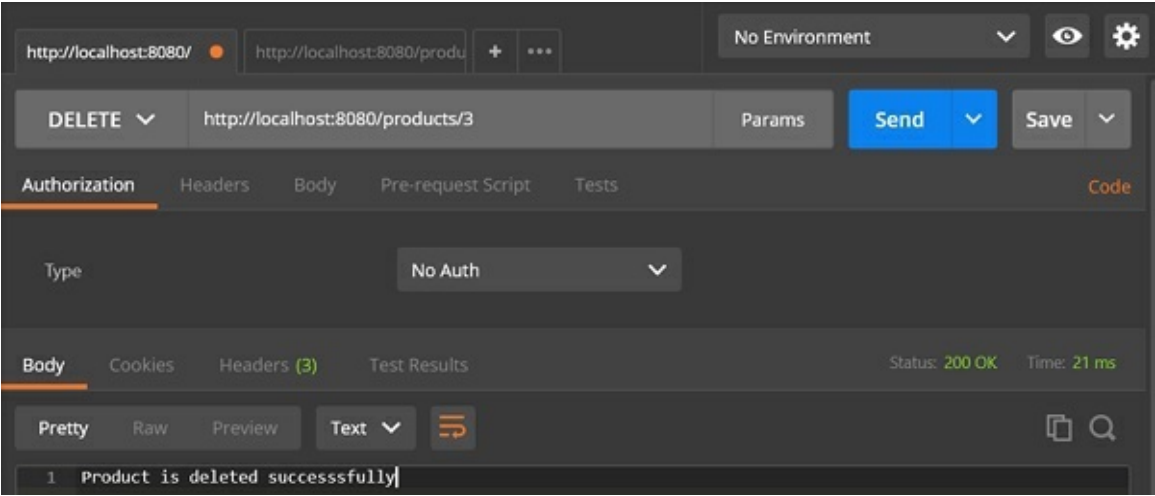
POST API URL is: <http://localhost:8080/products>



PUT API URL is: <http://localhost:8080/products/3>



DELETE API URL is: <http://localhost:8080/products/3>





## 12. Consuming RESTful Web Services

Maven – pom.xml file

```
<?xml version = "1.0" encoding = "UTF-8"?>  
<project xmlns = "http://maven.apache.org/POM/4.0.0"  
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation = "http://maven.apache.org/POM/4.0.0  
    http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

```
  <modelVersion>4.0.0</modelVersion>  
  <groupId>com.tutorialspoint</groupId>  
  <artifactId>demo</artifactId>  
  <version>0.0.1-SNAPSHOT</version>  
  <packaging>jar</packaging>  
  <name>demo</name>  
  <description>Demo project for Spring Boot</description>
```

```
  <parent>  
    <groupId>org.springframework.boot</groupId>  
    <artifactId>spring-boot-starter-parent</artifactId>  
    <version>1.5.8.RELEASE</version>  
    <relativePath />  
  </parent>
```

```
<properties>
```

```
<project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
<project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
<java.version>1.8</java.version>
</properties>
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>

  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
  </dependency>
</dependencies>
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.springframework.boot</groupId>
```

```
<artifactId>spring-boot-maven-plugin</artifactId>  
</plugin>  
</plugins>  
</build>  
  
</project>
```

Create a simple Spring Boot web application and write a controller class files which is used to redirects into the HTML file to consumes the RESTful web services.

We need to add the Spring Boot starter Thymeleaf and Web dependency in our build configuration file.

he controller class file given below – ViewController.java is given below –

```
package test.com;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ViewController {

    @RequestMapping("/view-products")
    public String viewProducts() {
        return "view-products";
    }

    @RequestMapping("/add-product")
    public String addProducts() {
        return "add-products";
    }
}
```

The view-products.html file is given below –

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset = "ISO-8859-1"/>
    <title>View Products</title>
    <script src = "https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>

    <script>
      $(document).ready(function() {
        $.getJSON("http://localhost:9090/products", function(result) {
          $.each(result, function(key,value) {
            $("#productsJson").append(value.id+" "+value.name+" ");
          });
        });
      });
    </script>
  </head>

  <body>
    <div id = "productsJson"> </div>
  </body>
</html>
```

The add-products.html file is given below –

```
<!DOCTYPE html>
<html>
  <head>
```

```
<meta charset = "ISO-8859-1" />
<title>Add Products</title>
<script src = "https://ajax.googleapis.com/ajax/libs/jquery/3.2.1/jquery.min.js"></script>
```

```
<script>
$(document).ready(function() {
    $("button").click(function() {
        var productmodel = {
            id : "3",
            name : "Ginger"
        };
        var requestJSON = JSON.stringify(productmodel);
        $.ajax({
            type : "POST",
            url : "http://localhost:9090/products",
            headers : {
                "Content-Type" : "application/json"
            },
            data : requestJSON,
            success : function(data) {
                alert(data);
            },
            error : function(data) {
            }
        });
    });
});
```

```
    </script>
</head>

<body>
    <button>Click here to submit the form</button>
</body>
</html>
```

The main Spring Boot Application class file is given below –

```
package com.tutorialspoint.demo;

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

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

Now, you can create an executable JAR file, and run the Spring Boot application by using the following Maven or Gradle commands.

For Maven, use the command as given below –

```
mvn clean install
```

After “BUILD SUCCESS”, you can find the JAR file under the target directory.

Run the JAR file by using the following command –

```
java -jar <JARFILE>
```

Now, the application has started on the Tomcat port 8080.

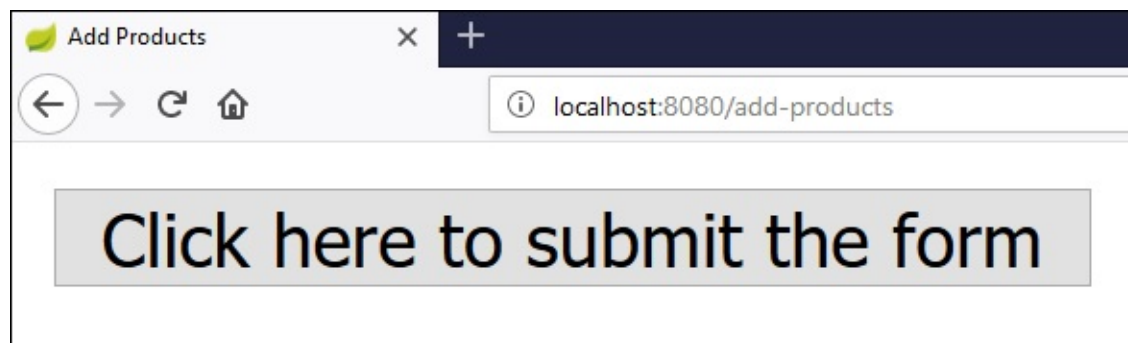
Now hit the URL in your web browser and you can see the output as shown –

<http://localhost:8080/view-products>



图七、1Honey\_2Almond

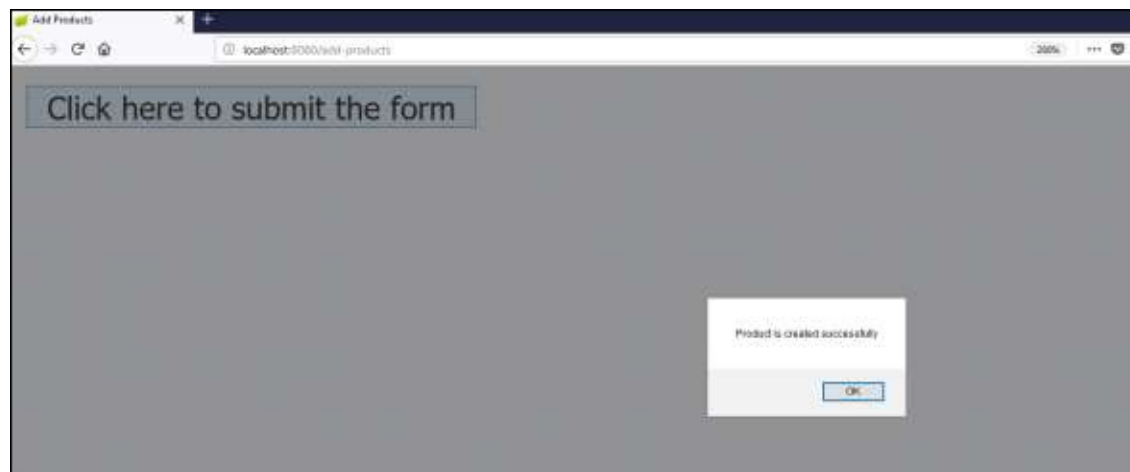
<http://localhost:8080/add-products>



图八、Submit Form Spring Boot



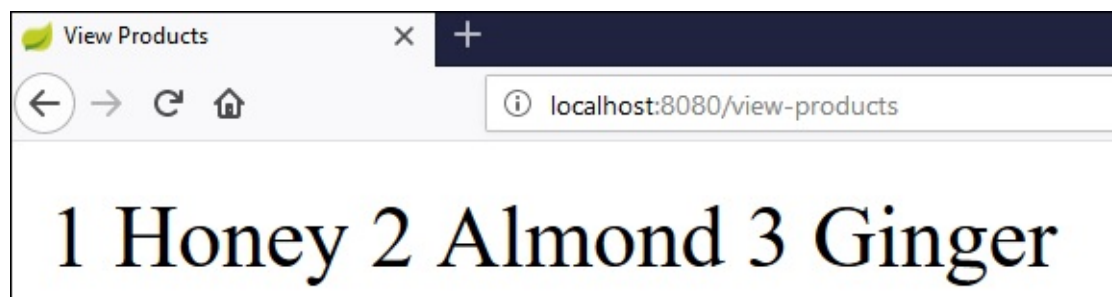
Now, click the button [Click here to submit the form](#) and you can see the result as shown –



图九、 Submit Form Spring Boot Output Window

Now, hit the view products URL and see the created product.

<http://localhost:8080/view-products>



图十、 1Honey 2Almond 3Ginger

## 12.1. Angular JS

To consume the APIs by using Angular JS, you can use the examples given below –

Use the following code to create the Angular JS Controller to consume the GET API - <http://localhost:9090/products> –

```
angular.module('demo', [])  
.controller('Hello', function($scope, $http) {  
    $http.get('http://localhost:9090/products').  
    then(function(response) {  
        $scope.products = response.data;  
    });  
});
```

Use the following code to create the Angular JS Controller to consume the POST API - <http://localhost:9090/products> –

```
angular.module('demo', [])  
.controller('Hello', function($scope, $http) {  
    $http.post('http://localhost:9090/products', data).  
    then(function(response) {  
        console.log("Product created successfully");  
    });  
});
```

**Note** – The Post method data represents the Request body in JSON format to create a product.

# Thank you

全国统一咨询热线：400-690-6115

北京|上海|广州|深圳|天津|成都|重庆|武汉|济南|青岛|杭州|西安

easthome.com