

# RESTfull Service with Spring Boot





# RESTfull Web Services

- A Web service is a modular application that you can describe, publish, locate, and invoke across the web. A Web service perform functions, which can be anything from simple requests to complicated business processes. Once a Web service is deployed, other applications or other Web services can discover and invoke the deployed service.
- REST defines a set of architectural principles by which you can design Web services that focus on a system's resources, including how resource states are addressed and transferred over HTTP by a wide range of clients written in different languages. If measured by the number of Web services that use it, REST has emerged in the last few years alone as a predominant Web service design model

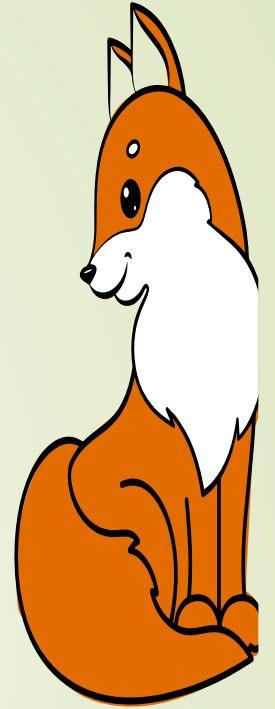


a concrete implementation of a REST Web service follows four basic design principles:

- Use HTTP methods explicitly.
- Be stateless.
- Expose directory structure-like URIs.
- Transfer XML, JavaScript Object Notation (JSON), or both.


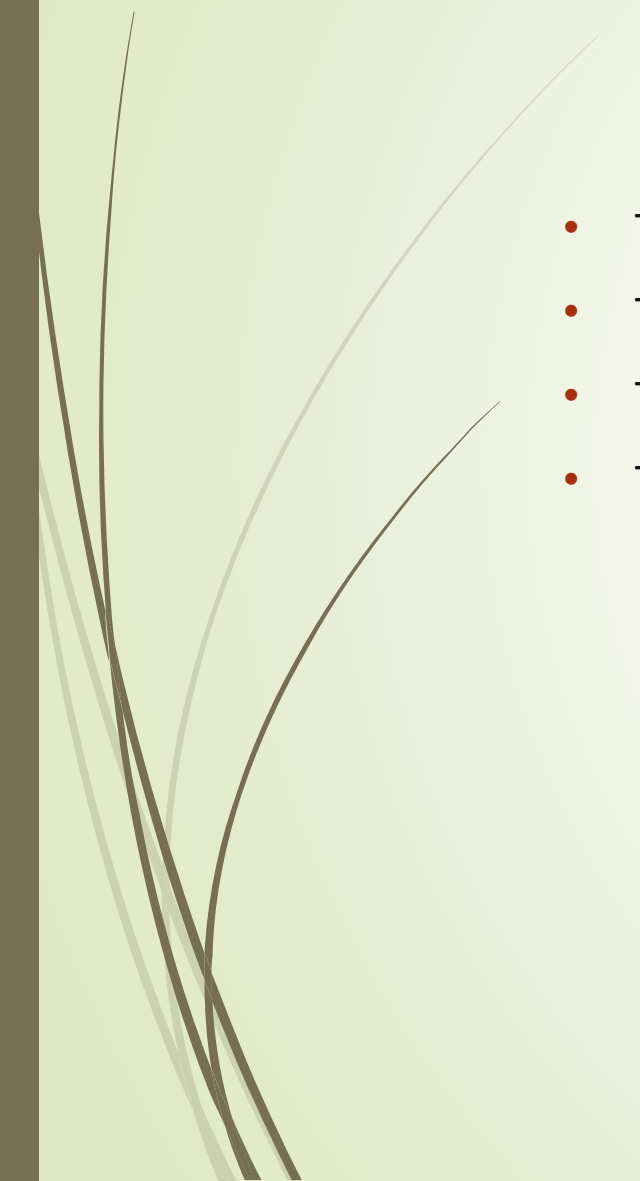
# Use HTTP methods explicitly

- One of the key characteristics of a RESTful Web service is the explicit use of HTTP methods in a way that follows the protocol as defined by RFC 2616. HTTP GET, for instance, is defined as a data-producing method that's intended to be used by a client application to retrieve a resource, to fetch data from a Web server, or to execute a query with the expectation that the Web server will look for and respond with a set of matching resources.



- ▶ REST asks developers to use HTTP methods explicitly and in a way that's consistent with the protocol definition. This basic REST design principle establishes a one-to-one mapping between create, read, update, and delete (CRUD) operations and HTTP methods. According to this mapping:



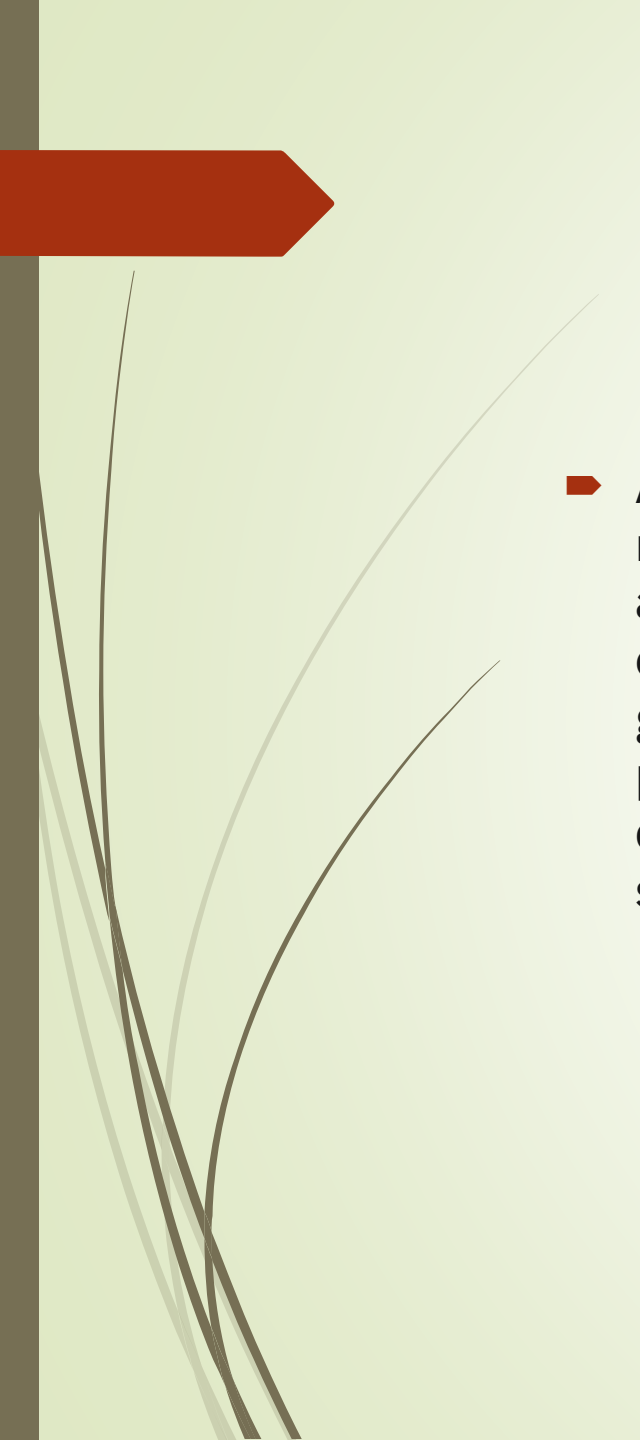
- 
- 
- To create a resource on the server, use POST.
  - To retrieve a resource, use GET.
  - To change the state of a resource or to update it, use PUT.
  - To remove or delete a resource, use DELETE.





# Be Stateless


- ▶ REST Web services need to scale to meet increasingly high performance demands. Clusters of servers with load-balancing and failover capabilities, proxies, and gateways are typically arranged in a way that forms a service topology, which allows requests to be forwarded from one server to the other as needed to decrease the overall response time of a Web service call. Using intermediary servers to improve scale requires REST Web service clients to send complete, independent requests; that is, to send requests that include all data needed to be fulfilled so that the components in the intermediary servers may forward, route, and load-balance without any state being held locally in between requests.


- 
- A complete, independent request doesn't require the server, while processing the request, to retrieve any kind of application context or state. A REST Web service application (or client) includes within the HTTP headers and body of a request all of the parameters, context, and data needed by the server-side component to generate a response. Statelessness in this sense improves Web service performance and simplifies the design and implementation of server-side components because the absence of state on the server removes the need to synchronize session data with an external application.





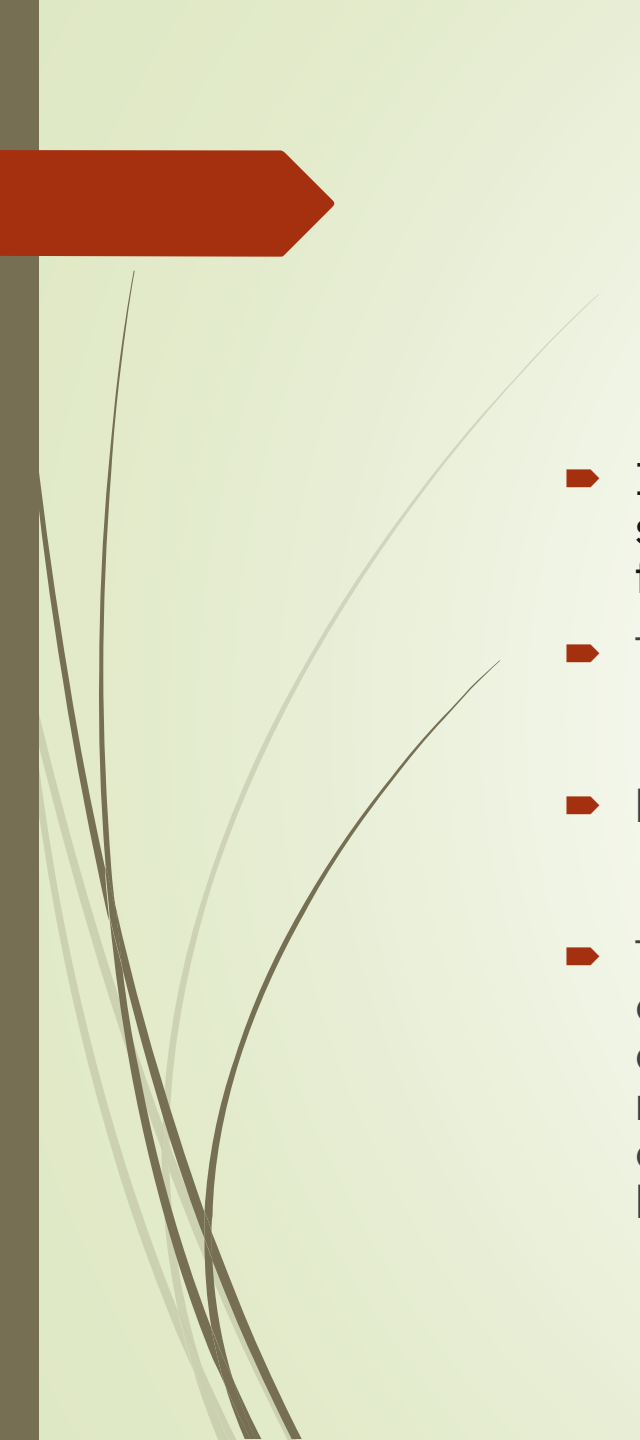



# Expose directory structure-like URIs

- From the standpoint of client applications addressing resources, the URIs determine how intuitive the REST Web service is going to be and whether the service is going to be used in ways that the designers can anticipate. A third RESTful Web service characteristic is all about the URIs.
- 

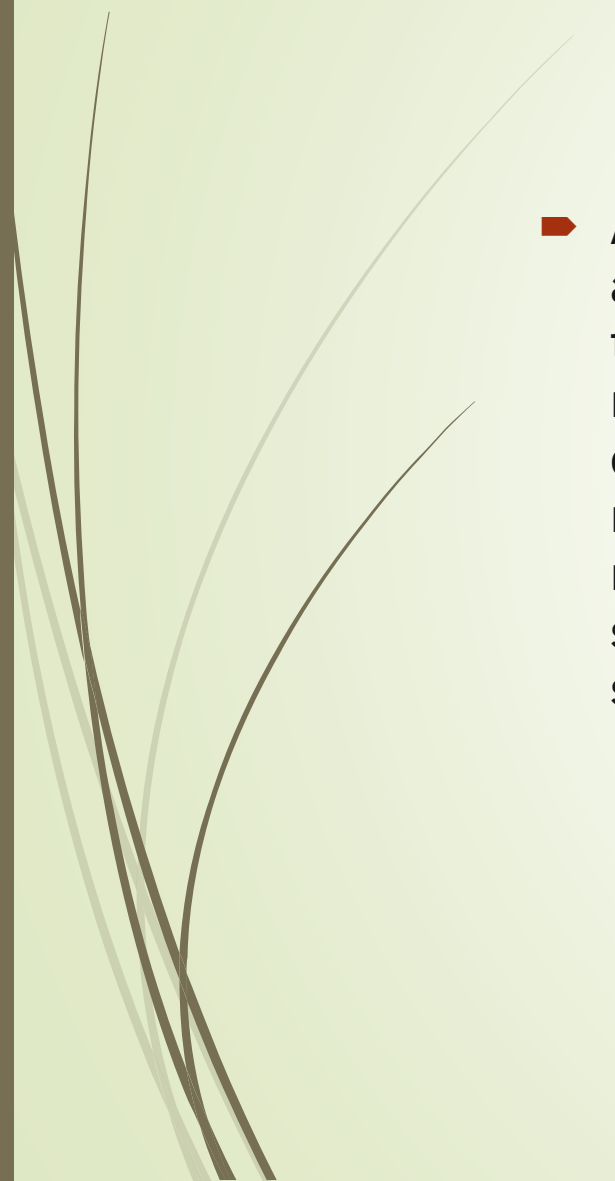
- 
- ▶ REST Web service URIs should be intuitive to the point where they are easy to guess. Think of a URI as a kind of self-documenting interface that requires little, if any, explanation or reference for a developer to understand what it points to and to derive related resources. To this end, the structure of a URI should be straightforward, predictable, and easily understood.



- 
- 
- One way to achieve this level of usability is to define directory structure-like URIs. This type of URI is hierarchical, rooted at a single path, and branching from it are subpaths that expose the service's main areas. According to this definition, a URI is not merely a slash-delimited string, but rather a tree with subordinate and superordinate branches connected at nodes. For example, in a discussion threading service that gathers topics ranging from Java to paper, you might define a structured set of URIs like this:
  - `http://www.myservice.org/discussion/topics/{topic}`
  - The root, `/discussion`, has a `/topics` node beneath it. Underneath that there are a series of topic names, such as `gossip`, `technology`, and so on, each of which points to a discussion thread. Within this structure, it's easy to pull up discussion threads just by typing something after `/topics/`.

- 
- In some cases, the path to a resource lends itself especially well to a directory-like structure. Take resources organized by date, for instance, which are a very good match for using a hierarchical syntax.
  - This example is intuitive because it is based on rules:
  - `http://www.myservice.org/discussion/2008/12/10/{topic}`
  - The first path fragment is a four-digit year, the second path fragment is a two-digit day, and the third fragment is a two-digit month. It may seem a little silly to explain it that way, but this is the level of simplicity we're after. Humans and machines can easily generate structured URIs like this because they are based on rules. Filling in the path parts in the slots of a syntax makes them good because there is a definite pattern from which to compose them:



# Transfer XML, JSON, or both

- 
- A resource representation typically reflects the current state of a resource, and its attributes, at the time a client application requests it. Resource representations in this sense are mere snapshots in time. This could be a thing as simple as a representation of a record in a database that consists of a mapping between column names and XML tags, where the element values in the XML contain the row values. Or, if the system has a data model, then according to this definition a resource representation is a snapshot of the attributes of one of the things in your system's data model. These are the things you want your REST Web service to serve up.

- 
- 
- The last set of constraints that goes into a RESTful Web service design has to do with the format of the data that the application and service exchange in the request/response payload or in the HTTP body. This is where it really pays to keep things simple, human-readable, and connected.
  - The objects in your data model are usually related in some way, and the relationships between data model objects (resources) should be reflected in the way they are represented for transfer to a client application. In the discussion threading service, an example of connected resource representations might include a root discussion topic and its attributes, and embed links to the responses given to that topic.





# Creating a RESTfull Web Service using Spring Boot

➤ A spring boot application contains a main method annotated with  
`@SpringBootApplication`

It is a combination of

`@ComponentScan`

`@Configuration`

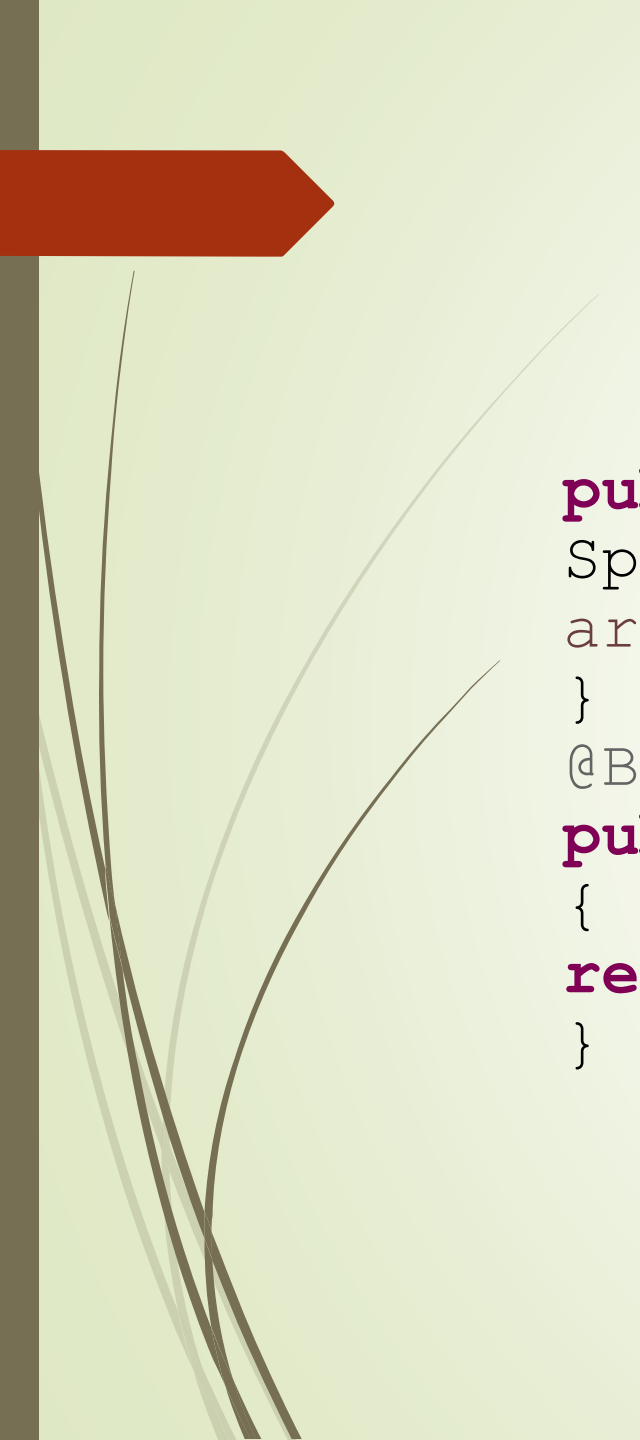
`@EnableAutoConfiguration`





# Creating a Bean

```
public class HelloWorldBean {  
    public String printMessage()  
    {  
        return "Hello World!Welcome To Spring Bean";  
    }  
  
}
```



```
public static void main(String[] args) {  
    SpringApplication.run(RestServiceApp.class,  
    args);  
}  
@Bean  
public HelloWorldBean getBean()  
{  
    return new HelloWorldBean();  
}
```



```
import
```

```
org.springframework.beans.factory.annotation.Autowired;
```

```
import
```

```
org.springframework.web.bind.annotation.GetMapping;
```

```
import
```

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

```
@RestController
```

```
public class HelloWorldController {
```

```
    @Autowired
```

```
    HelloWorldBean bean;
```

```
    @GetMapping("/helloworld")
```

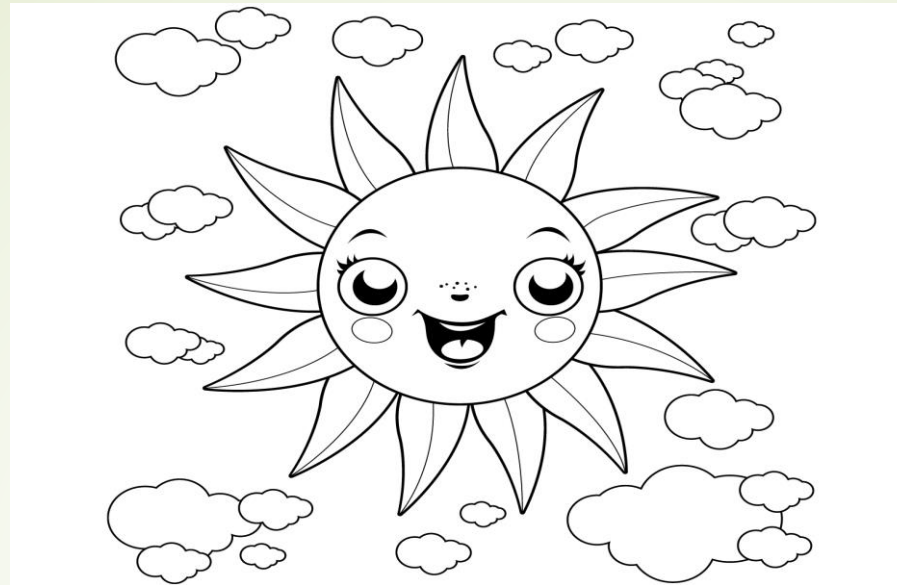
```
public String getHelloWorld()
```

```
{
```

```
    return bean.printMessage();
```

```
}
```

```
}
```



# Creating Model Classes ,@Service annotation and Using Http Methods