Spring Boot is a Java framework that is built on the top of the Spring Framework. It provides an easier and faster way to set up, configure, and run production-ready applications.

Need for Spring Boot

1) To use plain Spring Framework requires lot of configurations. If we want to use Spring MVC we need to use @ComponentScan annotation, Dispatcher Servlet, view resolver configuration and other jars. This kind of bootstrapping of applications will be time taking when building microservices.
2) A lot of manual configuration needs to be done to achieve production-ready application.
   a) Dependency Management: Configuration includes dependency management which developer must manage the required dependencies and their version compatibility.
   b) Configuration needed on web.xml
   c) Configuration like component scan, view resolver etc. in context.xml
   d) Take care of Non-Functional Requirements (NFR) like logging, Error Handling, Monitoring etc.

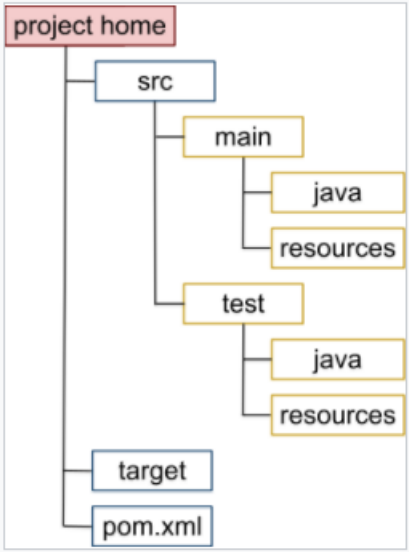
   Spring Boot is all about bootstrapping applications in an opinionated way with some default setting that can be overridden through configuration. Which means Spring Boot is a pre-configured spring platform.

Spring Boot is a convention over configuration extension for Spring Framework.

Convention over Configuration

Convention Over Configuration is a software design paradigm used by frameworks. It is used to reduce the configuration needed by the framework. Frameworks will follow a convention to achieve the tasks and any deviation from the convention will need a configuration.

For example, Maven uses this design paradigm, Maven follows the convention of having generic project directory structure of where to place the source files, resources, test files etc. Any deviation from this requires extra configuration.



Spring Boot Components

1) **Starter projects:**
   Spring boot has built-in starter projects which makes development easier and faster.

   Starter projects have pre-defined dependencies and configurations for each use-case.

   Starter projects are dependencies that can be included in the application.

   To build spring REST service, we must take care of appropriate dependencies, JSON conversion, Tomcat integration, Unit tests.

   It would be great if we can group all these under one single dependency. That's what starter projects do. They provide convenient descriptors for different use-cases.

   A Starter project can have other starter projects as dependencies.

Starter Project	**Description**
spring-boot-starter	It is used for core starter, including auto-configuration support, logging, and YAML.
spring-boot-starter-web	It is used for building the web application, including RESTful applications using Spring MVC. It uses Tomcat as the default embedded container.
spring-boot-starter-test	It is used to test Spring Boot applications with libraries, including JUnit, Hamcrest, and Mockito
spring-boot-starter-jdbc	It is used for JDBC with the Tomcat JDBC connection pool.
spring-boot-starter-data-jpa	It is used for Spring Data JPA with Hibernate.
spring-boot-starter-actuator	It is used for Spring Boot's Actuator that provides production-ready features to help you monitor and manage your application.
spring-boot-starter-web-services	It is used for development of SOAP web services using Spring Web Services (Spring-WS).

Use Case	**Starter Project needed**
REST and web apps	spring-boot-starter-web spring-boot-starter-test spring-boot-starter-web internally has dependencies for spring context, JSON conversion, Tomcat support, Spring Web, Spring MVC spring-web, spring-webmvc, spring-boot-starter-tomcat, spring boot-starter-json

Spring Auto Configuration

Spring Boot's auto-configuration is a powerful feature that automatically configures your Spring application based on the jar dependencies present in the classpath.

This eliminates the need for manual configuration of common components, leading to faster development and less boilerplate code.

Example:

Let's say you add the **spring-boot-starter-web** dependency to your project. Spring Boot will automatically: Configure a Tomcat web server, Setup a DispatcherServlet, Enable Spring MVC, JSON conversion and configure static resource handling.

You can override default auto-configuration behavior by setting properties in your application.properties or application.yml file.

If you have spring-boot-starter-tomcat on the classpath spring will automatically configures embedded tomcat with predefined configuration like 8080 for port etc. If you want to customize, define the property server.port on application.properties or application.yml file.

Dev Tools

Increase developer productivity.

Unit testing becomes easy.

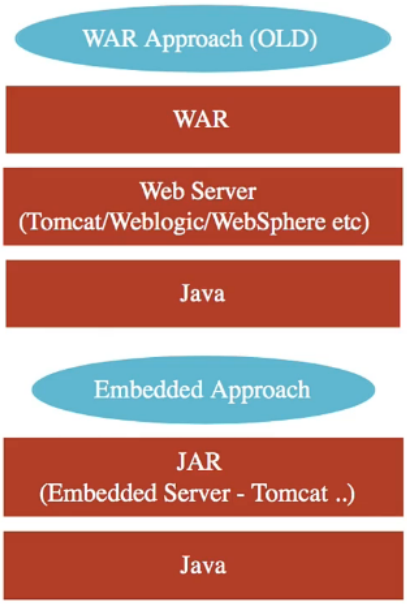
Why to restart the server when code is changed.

Starter project for this is spring-starter-devtools. Adding this starter project will allow to modify code without restarting the server.

Embedded Servers

Traditional war deployment approach is

1) Install Java
2) Install server (Tomcat/ JBoss/WebSphere etc.)
3) Deploy the war on to the server.



In Embedded server model, server is part of the Jar. We just need to install java and run it.

1) Install Java
2) Run Jar file which has tomcat embedded.

Actuator:

Actuator is helpful in application monitoring and management in production.

Actuator provides number of end points for application monitoring

1) End point for list of spring beans in your app.
2) Application health information
3) Application metrics
4) Request mapping details.

spring-boot-starter-actuator is the starter project to be included to include the actuator in your project.

http://localhost:8080/actuator

http://localhost:8080/actuator/health

include below property to expose endpoints

management.endpoints.web.exposure.include=*

http://localhost:8080/actuator/beans

http://localhost:8080/actuator/configprops

http://localhost:8080/actuator/env

http://localhost:8080/actuator/metrics

http://localhost:8080/actuator/metrics/http.server.requests

Instead of enabling all endpoints, we can selectively enable only for health and metrics like below

management.endpoints.web.exposure.include=health,metrics

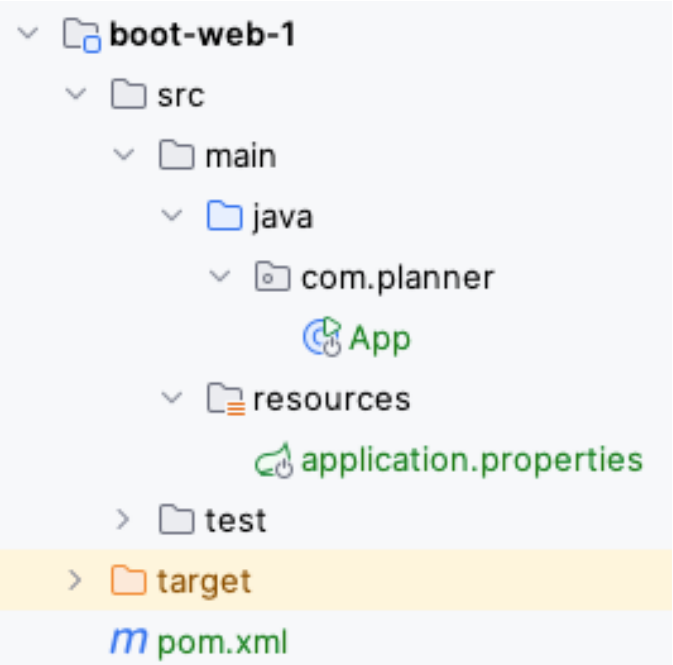
Web application in Spring Boot

Building web application is complex and need to consider below

1) Front end tech like JSP, JSTL, Bootstrap, HTML, CSS etc.
2) Framework to use like Spring MVC
3) Security like Spring security
4) Backend integration like Database using JPA, Kafka etc.

Steps to create sample app

1) Create a sample maven web application. Archetype can be "maven-archetype-quickstart".
2) Below directories will get created.



3) Add below entries in pom.xml and reload the project

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>3.4.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>3.4.0</version>
</dependency>
```

Spring-boot-starter-web internally has below dependencies

```xml
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter</artifactId>
  <version>3.4.0</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-json</artifactId>
  <version>3.4.0</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-tomcat</artifactId>
  <version>3.4.0</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-web</artifactId>
  <version>6.2.0</version>
  <scope>compile</scope>
</dependency>
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>6.2.0</version>
  <scope>compile</scope>
</dependency>
```

spring-boot-starter: This will add support for spring-core, auto-configuration, logging and YAML

spring-boot-starter-tomcat: Starter for using Tomcat as the embedded servlet container. Default servlet container starter used by spring-boot-starter-web

spring-webmvc: Adds support for spring context, beans, aop, expression, web.

Spring-boot-starter-json: Adds support for JSON reading, writing and conversion.

4) Add below annotations to the main class

```java
@SpringBootApplication  new *
public class App
{
    public static void main( String[] args )  nev
    {
        SpringApplication.run(App.class, args);
    }
}
```

SpringBootApplication annotation bootstraps spring boot application.

```java
@SpringBootApplication
public class BootDemoApplication {

    public static void main(String[] args) {
        SpringApplication.run(BootDemoApplication.cla
    }
}
```

```java
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@SpringBootConfiguration
@EnableAutoConfiguration
@ComponentScan(excludeFilters = {
        @Filter(type = FilterType.CUSTOM, classes = T
        @Filter(type = FilterType.CUSTOM, classes = A
public @interface SpringBootApplication {

    /**
     * Exclude specific auto-configuration classes su
     * @return the classes to exclude
     */
    @AliasFor(annotation = EnableAutoConfiguration.cl
    Class<?>[] exclude() default {};
```

Source code of SpringBootApplication class has annotations @ComponentScan that will scan all

the classes within the package where the class is defined and @EnableAutoConfiguration will automatically configuration application based on dependencies added.

REST Web Services

1) REST (Representational State

Transfer) is an architectural style for developing loosely coupled web-services. It is mainly useful to develop lightweight, fast, scalable and easy to maintain web services.

2) Business objects / data are

modeled as resources which are accessed using unique resource identifiers (uri). HTTP methods GET, PUT, POST, DELETE are used to perform action on those resources. These URI are called REST endpoints.

3) REST webservices follows

client-server architecture typically using Http Protocol.

REST BASIC CONSTRAINTS

There are six architectural constraints of Restful web-services.

Constraint	
Client-server	This constraint mandates the separation of concerns between the client side of the program and the server side of the program. Clients should not know how the server is fetching the resources, what database is used on the server side, what language has been used to develop the server-side program, and all other backend related stuff. Similarly, the server should not know how the client is using the resources it has requested, or how the user

	interface has been designed and any other frontend related stuff.
Uniform Interface	1) Identification of resources : This ensures the terms for the first sub-constraint for Uniform Interface constraint, i.e. Identification of Resources. This means that each resource must have it's unique identifiable URI. 2) Manipulation of resources through representations — The resources should have uniform representations in the server response. API consumers should use these representations to modify the resources state in the server. 3) **Self-descriptive messages** — Each resource representation should carry enough information to describe how to process the message. It should also provide information of the additional actions that the client can perform on the resource. The response comes with 'Content-Type' headers which tells the client that the resource returned is in json, or xml or plain text. It comes with 'Allow'

	headers which tells the client all the methods that are allowed to operate on the resource. Similarly, a lot of information is available on the header which makes it easy for the client to understand the Response and take appropriate action. 4) **Hypermedia as the engine of application state (HATEOAS)**— The client should have only the initial URI of the application. The client application should dynamically drive all other resources and interactions with the use of hyperlinks.
Layered system	This constraint tells that the architecture of the server can be layered, without letting the client know about it. For example, in a layered system, each component cannot see beyond the immediate layer they are interacting with. In a layered system architecture, the client can connect to other authorized intermediaries between the client and server, and it will still

	receive responses from the server. Servers can also pass on requests to other servers. You can design your RESTful web service to run on several servers with multiple layers such as security, application, and business logic, working together to fulfill client requests. These layers remain invisible to the client.
Cacheable	This constraint tells that the response from a server can be cacheable. With caching the response at the client side, the client does not need to request the server again and again to get the response. Downside is with cache in place, it needs to be updated each time the data is getting updated server side. There should be no difference in the response that can be retrieved directly from the server or the one from the cache. There are several headers that give information about caching, in both the request and response. Take a look here to get more details of the HTTP headers used for the providing information on caching.
Stateless	This constraint is focused on sending the state of the client each and every time it requests the server for a resource. A request should contain

	everything that is needed to fulfill the request, as the server need not store any session information about the client. The request might include the client credentials or session information or any token issued to the client to validate it, also any context information to understand the request, as the server won't also store anything from the previous client interactions. Server will validate the client request each and every time with the information sent in the request. The advantage with this is the server will be free of keeping the session information and managing it. It can process each request individually, and also no need of checking the previous interactions to understand the context.

Code on Demand	Although this functionality is optional,REST also allows client functionality to extend by downloading and executing code in the form of applets or scripts. The downloaded code simplifies clients by reducing the number of features required to be pre-implemented. Servers can provide part of features delivered to the client in the form of code, and the client only needs to execute the code.

SOAP	REST
SOAP is a protocol	REST is an architectural style
SOAP server and client applications are tightly coupled and bind with the WSDL contract	There is no contract in REST web services and client
SOAP works with XML only.	REST web services request and response types can be XML, JSON, text etc.
JAX-WS is the Java API for SOAP web services.	JAX-RS is the Java API for REST web services

	SOAP	REST
Stands for	Simple Object Access Protocol	Representational State Transfer
What is it?	SOAP is a protocol for communication between applications	REST is an architecture style for designing communication interfaces.
Design	SOAP API exposes the operation.	REST API exposes the data.

Transport Protocol	SOAP is independent and can work with any transport protocol.	REST works only with HTTPS.
Data format	SOAP supports only XML data exchange.	REST supports XML, JSON, plain text, HTML.
Performance	SOAP messages are larger, which makes communication slower.	REST has faster performance due to smaller messages and caching support.
Scalability	SOAP is difficult to scale. The server maintains state by storing all previous messages exchanged with a client.	REST is easy to scale. It's stateless, so every message is processed independently of previous messages.
Security	SOAP supports encryption with additional overheads.	REST supports encryption without affecting performance.
Use case	SOAP is useful in legacy applications and private APIs.	REST is useful in modern applications and public APIs.

SOAP	REST
• SOAP stands for Simple Object Access Protocol	• REST stands for Representational State Transfer
• SOAP is a protocol. SOAP was designed with a specification. It includes a WSDL file which has the required information on what the web service does in addition to the location of the web service.	• ○ REST is an Architectural style in which a web service can only be treated as a RESTful service if it follows the constraints of being • 1. Client Server 2. Stateless 3. Cacheable 4. Layered System 5. Uniform Interface
• SOAP cannot make use of REST since SOAP is a protocol and REST is an architectural pattern.	• REST can make use of SOAP as the underlying protocol for web services, because in the end it is just an architectural pattern.
• SOAP uses service interfaces to expose its functionality to client applications. In SOAP, the WSDL file provides the client with the necessary information which can be used to	• ○ REST use Uniform Service locators to access to the components on the hardware device. For example, if there is an object which represents the data of an employee hosted on a URL as http://demo.guru99 ,

understand what services the web service can offer.	the below are some of URI that can exist to access them http://demo.guru99.com/Employee http://demo.guru99.com/Employee/1
• SOAP requires more bandwidth for its usage. Since SOAP Messages contain a lot of information inside of it, the amount of data transfer using SOAP is generally a lot.	• ○ REST does not need much bandwidth when requests are sent to the server. REST messages mostly just consist of JSON messages. Below is an example of a JSON message passed to a web server. You can see that the size of the message is comparatively smaller to SOAP.

`<?xml version="1.0"?>`
`<SOAP-ENV:Envelope`
`xmlns:SOAP-ENV`
`="http://www.w3.org/2001/12/soap-envelope"`
`SOAP-ENV:encodingStyle`
`=" http://www.w3.org/2001/12/soap-encoding">`
`<soap:Body>`
`<Demo.guru99WebService`
`xmlns="http://tempuri.org/">`
`<EmployeeID>int</EmployeeID>`
`</Demo.guru99WebService>`
`</soap:Body>`
`</SOAP-ENV:Envelope>`
	{"city":"Mumbai","state":"Maharastra"}

• SOAP can only work with XML format. As seen from SOAP messages, all data passed is in XML format.	• REST permits different data format such as Plain text, HTML, XML, JSON, etc. But the most preferred format for transferring data is JSON.

When to use REST and when to use SOAP

One of the most highly debatable topics is when REST should be used or when to use SOAP while designing web services.

Below are some of the key factors that determine when each technology should be used for web services **REST services should be used in the following instances**

- **Limited resources and bandwidth** – Since SOAP messages are heavier in content and consume a far greater bandwidth, REST should be used in instances where network bandwidth is a constraint.
- **Statelessness** – If there is no need to maintain a state of information from one request to another then REST should be used. If you need a proper information flow wherein some information from one request needs to flow into another then SOAP is more suited for that purpose. We can take the example of any online purchasing site. These sites normally need the user first to add items which need to be purchased to a cart. All of the cart items are then transferred to the payment page in order to complete the purchase. This is an

example of an application which needs the state feature. The state of the cart items needs to be transferred to the payment page for further processing.

- **Caching** – If there is a need to cache a lot of requests then REST is the perfect solution. At times, clients could request for the same resource multiple times. This can increase the number of requests which are sent to the server. By implementing a cache, the most frequent queries results can be stored in an intermediate location. So whenever the client requests for a resource, it will first check the cache. If the resources exist then, it will not proceed to the server. So caching can help in minimizing the amount of trips which are made to the web server.
- **Ease of coding** – Coding REST Services and subsequent implementation is far easier than SOAP. So if a quick win solution is required for web services, then REST is the way to go.

SOAP should be used in the following instances

1. **Asynchronous processing and subsequent invocation** – if there is a requirement that the client needs a guaranteed level of reliability and security then the new SOAP standard of SOAP 1.2 provides a lot of additional features, especially when it comes to security.
2. **A Formal means of communication** – if both the client and server have an agreement on the exchange format then SOAP 1.2 gives the rigid specifications for this type of interaction. An example is an online purchasing site in which users add items to a cart before the payment is made. Let's assume we have a web service that does the final payment. There can be a firm agreement that the web service will only accept the cart item name, unit price, and quantity. If such a scenario exists then, it's always better to use the SOAP protocol.
3. **Stateful operations –** if the application has a requirement that state needs to be maintained from one request to another, then the SOAP 1.2 standard provides the WS* structure to support such requirements.