What is Micro Service?

* A small, focussed piece of software
* Independently developed,deployed,upgraded
* Commonly exposes it functionality via HTTP/REST

The micro service architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with light weight mechanism, often an HTTP resource API.

**Micro services - Approach to develop small yet capable software**

* Do one thing well
* Play well with other programs
* Use standard interfaces

**Monoliths - Built and deployed as single artefact**

* Easy to setup – single project in IDE
* Easy to deploy – single WAR file
* Scaled horizontally
* [Uniform Interface](http://www.restapitutorial.com/lessons/whatisrest.html)
* [Stateless](http://www.restapitutorial.com/lessons/whatisrest.html)
* [Cacheable](http://www.restapitutorial.com/lessons/whatisrest.html)
* [Client-Server](http://www.restapitutorial.com/lessons/whatisrest.html)
* [Layered System](http://www.restapitutorial.com/lessons/whatisrest.html)
* [Code on Demand (optional)](http://www.restapitutorial.com/lessons/whatisrest.html)

Spring boot is sub-project developed by developers of spring framework – to create stand-alone, production-grade application with minimum configuration possible. Spring boot applications are typically bundled as [fat/uber jar](http://howtodoinjava.com/maven/maven-shade-plugin-create-uberfat-jar-example/) files and can be deployed in any platform as a simple jar file.

Whatz happening?

Spring Boot uses convention over configuration by scanning the dependent libraries available in the class path. For each spring-boot-starter-\* dependency in the POM file, Spring Boot executes a default AutoConfiguration class. AutoConfiguration classes use the \*AutoConfiguration lexical pattern, where \* represents the library. For example, the autoconfiguration of spring security is done through SecurityAutoConfiguration.

At the same time, if you don’t want to use auto configuration for any project, it makes it very simple. Just use exclude = SecurityAutoConfiguration.class like below.

|  |
| --- |
| @SpringBootApplication (exclude = SecurityAutoConfiguration.class)  public class SpringBootDemoApplication {      public static void main(String[] args) {          SpringApplication.run(SpringBootDemoApplication.class, args);      }  } |

It is also possible to override default configuration values using the application.properties file in src/main/resources folder.

Now look at @SpringBootApplication annotation what it actually does.

#### @SpringBootApplication Annotation

SpringBootApplication is defined as below:

|  |
| --- |
| @Target(ElementType.TYPE)  @Retention(RetentionPolicy.RUNTIME)  @Documented  @Inherited  @SpringBootConfiguration  @EnableAutoConfiguration  @ComponentScan(excludeFilters = @Filter(type = FilterType.CUSTOM, classes = TypeExcludeFilter.class))  public @interface SpringBootApplication  {      //more code  } |

It adds 3 important annotations for application configuration purpose.

#### @SpringBootConfiguration

|  |
| --- |
| @Configuration  public @interface SpringBootConfiguration  {      //more code  } |

1. This annotation adds @Configuration annotation to class which **mark the class a source of bean definitions for the application context.**

#### @EnableAutoConfiguration

This tells spring boot to auto configure important bean definitions based on added dependencies in pom.xmlby start adding beans based on classpath settings, other beans, and various property settings.

#### @ComponentScan

This annotation tells spring boot to scan base package, find other beans/components and configure them as well.

 To know what all beans have been automatically configured into your **spring boot**  then use this code and run it.

package com.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.context.ApplicationContext;

import java.util.Arrays;

@SpringBootApplication

public class DemoApplication {

public static void main(String[] args) {

//SpringApplication.run(DemoApplication.class, args);

ApplicationContext ctx = SpringApplication.run(DemoApplication.class, args);

String[] beanNames = ctx.getBeanDefinitionNames();

Arrays.sort(beanNames);

for (String beanName : beanNames)

{

System.out.println(beanName);

}

}

}

With my pom.xml file, it generates following beans names along with plenty of other springframework.boot.autoconfigure dependencies.

\_halObjectMapper

\_linkDiscovererRegistry

\_relProvider

alpsController

alpsConverter

alpsJsonHttpMessageConverter

annotatedEventHandlerInvoker

annotationRelProvider

associationLinks

auditableBeanWrapperFactory

backendIdConverterRegistry

backendIdHandlerMethodArgumentResolver

baseUri

basicErrorController

beanNameHandlerMapping

beanNameViewResolver

characterEncodingFilter

config

controllerEntityLinks

conventionErrorViewResolver

defaultConversionService

defaultMessageConverters

defaultRelProvider

defaultServletHandlerMapping

defaultViewResolver

delegatingEntityLinks

demoApplication

dispatcherServlet

dispatcherServletRegistration

domainObjectMerger

duplicateServerPropertiesDetector

eTagArgumentResolver

embeddedServletContainerCustomizerBeanPostProcessor

entityLinks

entityLinksPluginRegistry

enumTranslator

error

errorAttributes

errorPageCustomizer

errorPageRegistrarBeanPostProcessor

excerptProjector

faviconHandlerMapping

faviconRequestHandler

halJacksonHttpMessageConverter

halMessageConverterSupportedMediaTypeCustomizer

halObjectMapper

handlerExceptionResolver

hiddenHttpMethodFilter

httpHeadersPreparer

httpPutFormContentFilter

httpRequestHandlerAdapter

jacksonGeoModule

jacksonHttpMessageConverter

jacksonObjectMapper

jacksonObjectMapperBuilder

jpaHelper

jsonComponentModule

jsonSchemaConverter

jsr303Validator

linkCollector

linkRelationMessageSource

localeCharsetMappingsCustomizer

mappingJackson2HttpMessageConverter

mbeanExporter

mbeanServer

messageConverters

metadataConfiguration

methodValidationPostProcessor

multipartConfigElement

multipartResolver

mvcContentNegotiationManager

mvcConversionService

mvcPathMatcher

mvcResourceUrlProvider

mvcUriComponentsContributor

mvcUrlPathHelper

mvcValidator

mvcViewResolver

objectMapper

objectNamingStrategy

org.springframework.boot.autoconfigure.AutoConfigurationPackages

org.springframework.boot.autoconfigure.condition.BeanTypeRegistry

org.springframework.boot.autoconfigure.context.ConfigurationPropertiesAutoConfiguration

org.springframework.boot.autoconfigure.context.PropertyPlaceholderAutoConfiguration

org.springframework.boot.autoconfigure.dao.PersistenceExceptionTranslationAutoConfiguration

org.springframework.boot.autoconfigure.data.rest.RepositoryRestMvcAutoConfiguration

org.springframework.boot.autoconfigure.hateoas.HypermediaAutoConfiguration

org.springframework.boot.autoconfigure.hateoas.HypermediaHttpMessageConverterConfiguration

org.springframework.boot.autoconfigure.info.ProjectInfoAutoConfiguration

org.springframework.boot.autoconfigure.internalCachingMetadataReaderFactory

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$Jackson2ObjectMapperBuilderCustomizerConfiguration

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperBuilderConfiguration

org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration$JacksonObjectMapperConfiguration

org.springframework.boot.autoconfigure.jmx.JmxAutoConfiguration

org.springframework.boot.autoconfigure.transaction.TransactionAutoConfiguration

org.springframework.boot.autoconfigure.validation.ValidationAutoConfiguration

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletConfiguration

org.springframework.boot.autoconfigure.web.DispatcherServletAutoConfiguration$DispatcherServletRegistrationConfiguration

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration

org.springframework.boot.autoconfigure.web.EmbeddedServletContainerAutoConfiguration$EmbeddedTomcat

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$DefaultErrorViewResolverConfiguration

org.springframework.boot.autoconfigure.web.ErrorMvcAutoConfiguration$WhitelabelErrorViewConfiguration

org.springframework.boot.autoconfigure.web.HttpEncodingAutoConfiguration

org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration

org.springframework.boot.autoconfigure.web.HttpMessageConvertersAutoConfiguration$StringHttpMessageConverterConfiguration

org.springframework.boot.autoconfigure.web.JacksonHttpMessageConvertersConfiguration

org.springframework.boot.autoconfigure.web.JacksonHttpMessageConvertersConfiguration$MappingJackson2HttpMessageConverterConfiguration

org.springframework.boot.autoconfigure.web.MultipartAutoConfiguration

org.springframework.boot.autoconfigure.web.ServerPropertiesAutoConfiguration

org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration

org.springframework.boot.autoconfigure.web.WebClientAutoConfiguration$RestTemplateConfiguration

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$EnableWebMvcConfiguration

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter

org.springframework.boot.autoconfigure.web.WebMvcAutoConfiguration$WebMvcAutoConfigurationAdapter$FaviconConfiguration

org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration

org.springframework.boot.autoconfigure.websocket.WebSocketAutoConfiguration$TomcatWebSocketConfiguration

org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor

org.springframework.boot.context.properties.ConfigurationPropertiesBindingPostProcessor.store

org.springframework.context.annotation.internalAutowiredAnnotationProcessor

org.springframework.context.annotation.internalCommonAnnotationProcessor

org.springframework.context.annotation.internalConfigurationAnnotationProcessor

org.springframework.context.annotation.internalRequiredAnnotationProcessor

org.springframework.context.event.internalEventListenerFactory

org.springframework.context.event.internalEventListenerProcessor

org.springframework.data.rest.webmvc.config.RepositoryRestMvcConfiguration

org.springframework.data.web.config.SpringDataJacksonConfiguration

org.springframework.hateoas.LinkDiscoverers#0

org.springframework.hateoas.config.HateoasConfiguration

org.springframework.hateoas.config.HypermediaSupportBeanDefinitionRegistrar$DefaultObjectMapperCustomizer#0

org.springframework.hateoas.config.HypermediaSupportBeanDefinitionRegistrar$Jackson2ModuleRegisteringBeanPostProcessor#0

pageableResolver

pagedResourcesAssembler

pagedResourcesAssemblerArgumentResolver

persistenceExceptionTranslationPostProcessor

persistentEntities

persistentEntityArgumentResolver

platformTransactionManagerCustomizers

preserveErrorControllerTargetClassPostProcessor

profileController

profileResourceProcessor

projectionDefinitionRegistrar

propertySourcesPlaceholderConfigurer

relProviderPluginRegistry

repoRequestArgumentResolver

repositories

repositoryController

repositoryEntityController

repositoryExporterHandlerAdapter

repositoryInvokerFactory

repositoryPropertyReferenceController

repositoryRelProvider

repositoryRestExceptionHandler

repositorySchemaController

repositorySearchController

requestContextFilter

requestMappingHandlerAdapter

requestMappingHandlerMapping

resourceDescriptionMessageSourceAccessor

resourceHandlerMapping

resourceMappings

resourceMetadataHandlerMethodArgumentResolver

resourceProcessorInvoker

restHandlerMapping

restTemplateBuilder

selfLinkProvider

serverHttpRequestMethodArgumentResolver

serverProperties

simpleControllerHandlerAdapter

sortResolver

spring.data.rest-org.springframework.boot.autoconfigure.data.rest.RepositoryRestProperties

spring.hateoas-org.springframework.boot.autoconfigure.hateoas.HateoasProperties

spring.http.encoding-org.springframework.boot.autoconfigure.web.HttpEncodingProperties

spring.http.multipart-org.springframework.boot.autoconfigure.web.MultipartProperties

spring.info-org.springframework.boot.autoconfigure.info.ProjectInfoProperties

spring.jackson-org.springframework.boot.autoconfigure.jackson.JacksonProperties

spring.mvc-org.springframework.boot.autoconfigure.web.WebMvcProperties

spring.resources-org.springframework.boot.autoconfigure.web.ResourceProperties

spring.transaction-org.springframework.boot.autoconfigure.transaction.TransactionProperties

springBootRepositoryRestConfigurer

standardJacksonObjectMapperBuilderCustomizer

stringHttpMessageConverter

tomcatEmbeddedServletContainerFactory

uriListHttpMessageConverter

validatingRepositoryEventListener

viewControllerHandlerMapping

viewResolver

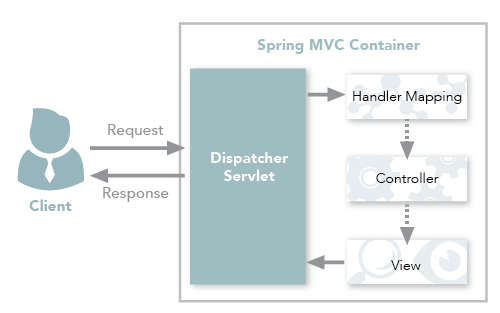
websocketContainerCustomizer

welcomePageHandlerMapping

**Spring Framework: @RestController vs. @Controller**

Spring’s annotation-based MVC framework simplifies the process of creating RESTful web services. The key difference between a traditional Spring MVC controller and the RESTful web service controller is the way the HTTP response body is created. While the traditional MVC controller relies on the View technology, the RESTful web service controller simply returns the object and the object data is written directly to the HTTP response as JSON/XML.

Figure 1 - Spring MVC traditional workflow



### Spring MVC REST Workflow

The following steps describe a typical Spring MVC REST workflow:

1. The client sends a request to a web service in URI form.
2. The request is intercepted by the DispatcherServlet which looks for Handler Mappings and its type.
   * The Handler Mappings section defined in the application context file tells DispatcherServlet which strategy to use to find controllers based on the incoming request.
   * Spring MVC supports three different types of mapping request URIs to controllers: annotation, name conventions, and explicit mappings.
3. Requests are processed by the Controller and the response is returned to the DispatcherServlet which then dispatches to the view.

In Figure 1, notice that in the traditional workflow the ModelAndView object is forwarded from the controller to the client. Spring lets you return data directly from the controller, without looking for a view, using the @ResponseBody annotation on a method.

## Using the @ResponseBody Annotation

When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the http response automatically. Each method in the Controller class must be annotated with @ResponseBody.

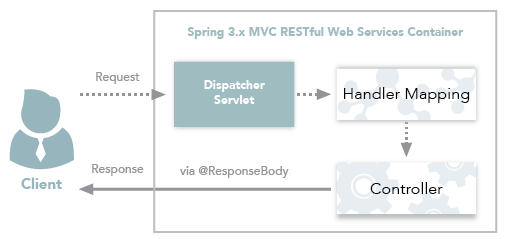


Figure 2: Spring 3.x MVC RESTful web services workflow

### Behind the Scenes

Spring has a list of HttpMessageConverters registered in the background. The responsibility of the HTTPMessageConverter is to convert the request body to a specific class and back to the response body again, depending on a predefined mime type. Every time an issued request hits @ResponseBody, Spring loops through all registered HTTPMessageConverters seeking the first that fits the given mime type and class, and then uses it for the actual conversion.

@ResponseBody with a simple example

package com.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

import org.springframework.web.bind.annotation.\*;

import org.springframework.stereotype.Controller;

import org.springframework.web.bind.annotation.PathVariable;

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

import org.springframework.web.bind.annotation.RequestMethod;

import org.springframework.web.bind.annotation.ResponseBody;

@Controller

@SpringBootApplication

public class DemoApplication {

@RequestMapping(value = "/", method = RequestMethod.GET, produces = "application/json")

public @ResponseBody String sayhello()

{

return "Hello @Controller";

}

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}

## Using the @RestController Annotation

Spring 4.0 introduced @RestController, a specialized version of the controller which is a convenience annotation that does nothing more than add the @Controller and @ResponseBody annotations. By annotating the controller class with @RestController annotation, you no longer need to add @ResponseBody to all the request mapping methods. The @ResponseBody annotation is active by default. 

To use @RestController in our example, all we need to do is modify the @Controller to @RestController and remove the @ResponseBody from each method.

package com.example;

import org.springframework.boot.SpringApplication;

import org.springframework.boot.autoconfigure.SpringBootApplication;

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

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

@RestController

@SpringBootApplication

public class DemoApplication {

@RequestMapping(value = "/")

public String sayhello()

{

return "Hello @RestController";

}

public static void main(String[] args) {

SpringApplication.run(DemoApplication.class, args);

}

}