**q)why nimbus?**

**Nimbus Platform reduces application development time by**

* Providing the ability to build application through configuration.
* Providing boilerplate code for cross cutting concerns.

--

Key features of the platform:

* Customized Workflow
* Configurable UI
* Multi Tenancy
* Cloud based solution

Useful for nimbus training:

**What does**@RequestMapping**annotation do?**([answer](http://javarevisited.blogspot.sg/2017/06/how-spring-mvc-framework-works-web-flow.html#axzz55vF5ugU8))

The @RequestMapping annotation is used to map web requests to Spring Controller methods. You can map request based upon HTTP methods  e.g. GET and POST and various other parameters. For examples, if you are developing RESTful Web Service using Spring then you can use produces and consumes property along with media type annotation to indicate that this method is only used to produce or consumers JSON as shown below:

|  |  |
| --- | --- |
| 1 | @RequestMapping (method = RequestMethod.POST, consumes="application/json") |
| 2 | public Book save(@RequestBody Book aBook) { |

|  |  |
| --- | --- |
| 3 | return bookRepository.save(aBook); |
| 4 | }  **What is the difference between**@Controller**and**@RestController**?** |

@RestController is a specialization of @Controller for RESTful web service. It not only combines  @ResponseBody and @Controller annotation but also gives more meaning to your controller class to clearly indicate that it deals with RESTful requests.

**Is REST normally stateless?**([answer](http://javarevisited.blogspot.sg/2015/08/difference-between-soap-and-restfull-webservice-java.html))

Yes, REST API should be stateless because it is based on HTTP which is also stateless. A Request in REST API should contain all the details required it to process i.e. it should not rely on previous or next request or some data maintained at the server end e.g. Sessions. REST specification put a constraint to make it stateless and you should keep that in mind while

designing your REST API.

--

**model** :the model can supply attributes used for rendering views.

**Autowiring** is method of creating an instance of an object and "by concept" injecting that instance on a specific class that uses it.

**ApplicationContext provides:**

* Bean factory methods for accessing application components. (Fancy talk for things you need in your application without using new() called [dependency injection](https://en.wikipedia.org/wiki/Dependency_injection))
* The ability to load file resources in a generic fashion. (External configuration with [properties](https://docs.spring.io/spring/docs/4.0.3.RELEASE/javadoc-api/org/springframework/context/annotation/PropertySource.html)and [profiles](https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-profiles.html))
* The ability to publish events to registered listeners. (Fancy talk for "when certain things happen do something", read about the [observer pattern](https://en.wikipedia.org/wiki/Observer_pattern))
* The ability to resolve messages to support internationalization. (Files that have all the Strings used in your application so they can be rendered in different languages, read about [MessageSource](http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/MessageSource.html))
* Inheritance from a parent context. (Like I said before, your application needs "context" and this is where it begins.)

As you are using Spring boot, there is only one context by default: ApplicationContext. This will contain all your things (Beans) and Components you need.

*@SpringBootApplication*

*@ComponentScan*(basePackages = "com.mydevgeek")

*@EnableJpaRepositories*(basePackages = "com.mydevgeek.repo")

*@EntityScan*(basePackages = "com.mydevgeek.domain")

public class Application {

    public static void main(String[] args) {

        SpringApplication.run(Application.class, args);

    }

}

@ComponentScan – use for scanning the components. In this example, controllers and services can not be identified, if we don’t define the component scan path.

--

@SpringBootApplication – This annotation automatically detects all the @Entity, @Repository, @Controller/ @RestController, @Service and @Component. But all components must be in same or child package level

--the Client sends a GET request to /foos with the Accept header set to application/json – to get all Foo resources as Json

• the Foo Spring Controller is hit and returns the corresponding Foo Java entities

• Spring then uses

--

**NOTE:**WebApplicationInitializer and ApplicationContextInitializer serve fairly different purposes. Whereas the WebApplicationInitializer is used by a Servlet Container at startup of the web application and provides a way for programmatic creating a web application(replacement for a web.xml file), ApplicationContextInitializer provides a hook to configure the Spring application context before it gets fully created.

There is always exactly one per webapp instance in a servlet container like Tomcat. It is not a part of Spring. You seldom use it directly when using Spring. But it is there in the background.

**Servlet Context**, d meaning of [ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html) class, is d application-wide context a Servlet webapp has.

In terms of system design, any Context is a collection of common functionality and data, that defined in one place, but has to be used from anywhere in program.

purpose of Context - *simplify access to common functionality and data*. Lets consider contexts (..)

But normally most applications have a single context that contains all the beans and components that your application requires.

For example, if you loaded an application using a Servlet you could load the entirety with a "root context" that also loads the Servlet Context as it's child.

**ServletContext**

ServletContext is part of [Servlet technology](http://docs.oracle.com/javaee/6/tutorial/doc/bnafd.html). Many of frameworks are based on this technology (*JSF, Spring, Struts and many more*). ServletContext contains functionality for communicate with its Servlet container(*like Tomcat, Glassfish etc*). Basic things that ServletContext provides:

* get application Initial parameters;
* get information about request dispatching;
* add or remove Servlets, Filters and Listeners;
* get or set container's attributes;
* log run-time messages;
* get additional information like Application path, Container version etc.

As you see there is only basic functionality, that can be useful anywhere in application, if you will work with Servlet technology (for example, develop another one mvc-framework based on it).

**ApplicationContext**

ApplicationContext is main interface of [Spring framework](http://docs.spring.io/spring/docs/current/spring-framework-reference/htmlsingle/) application. It has lot of implementations. One of them loads configuration from xml-file (*ClassPathXmlApplicationContext*), another one loads configuration based on annotations (*AnnotationConfigApplicationContext*) and so on. Basic things that ApplicationContext provides:

* bean factory, ability to create new beans based on bean definitions;
* load resources from different sources (file system, jar files, url etc);
* ability to publish and listen events;
* resolve messages from message bundles;

Again, initialized on start, this basic functionality can be useful almost anywhere in your application. That is why it collected to context. Many classes used in background, to provide this abilities, but all your have to know, for use this functionality - just ApplicationContext.

**You can create ApplicationContext instances declaratively** by using, for example, a ContextLoader. i.e You can **register an ApplicationContext using the ContextLoaderListener** .

Of course you can also create ApplicationContext instances programmatically by using one of the ApplicationContext implementations.

**SecurityContext**

SecurityContext provides access to authentication data. You can get the name of authenticated user, roles and other details. This information initialized by security module, may be needed in many many places. Сomponents, which use this information know nothing about classes of security module. They just get all needed information from SecurityContext.

There can be different interpretations, but here is how I see it:

* **Spring Security context**, in the meaning of [SecurityContext](http://docs.spring.io/spring-security/site/docs/current/apidocs/org/springframework/security/core/context/SecurityContext.html) class, holds the authentication, username, authorities (roles) and possibly other information about the current user. The lifespan of such context is the current request, or the security context is persisted between requests using sessions.
* **Spring Context**, in the meaning of [ApplicationContext](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/context/ApplicationContext.html) class, is the central point of a running Spring application. Its main purpose is to contain the app singleton beans, but it has many other nifty features (several mentioned in another answer). An application can have several ApplicationContexts, but the most common, and simplest case, it only has one. Web applications usually use the [WebApplicationContext](https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/context/WebApplicationContext.html) "extension", which integrates it with the Servlet context.

Difference between ApplicationContext and WebApplicationContext in Spring MVC

There are two types of contexts we are dealing with:

1. **root context** (parent)
2. **individual servlet context** (child)

if you are developing a Spring MVC web application you will typically have a  **rootWebApplicationContext loaded via Spring's ContextLoaderListener** and a **child WebApplicationContext loaded via Spring's DispatcherServlet**. This results in a parent-child context hierarchy where shared components and infrastructure configuration are declared in the root context and consumed in the child context by web-specific components.

The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications. It differs from a normal ApplicationContext in that it is capable of resolving themes (see Using themes), and that it knows which Servlet it is associated with (by having a link to the ServletContext). The WebApplicationContext is bound in the ServletContext, and by using static methods on the RequestContextUtils class you can always look up the WebApplicationContext if you need access to it.

Web Application context extended Application Context which is designed to work with the standard [javax.servlet.ServletContext](http://docs.oracle.com/javaee/6/api/javax/servlet/ServletContext.html) so it's able to communicate with the container.

Beans, instantiated in WebApplicationContext will also be able to use ServletContext if they implement ServletContextAware interface

The ApplicationContext and WebApplicationContext both are almost same thing but there are some basic differences related to the web aware environment. In Spring ApplicationContext instances can be scoped.

In the Web MVC framework, each DispatcherServlet has its own WebApplicationContext (i.e own \*-servlet.xml), which inherits all the beans already defined in the root WebApplicationContext. You can also override the inherited bean scope in the servlet-specific scope and also can define new scope-specific beans local to a given servlet instance.

So we can say that both ApplicationContext and WebApplicationContext are the spring containers where WebApplicationContext is child(extends) of the ApplicationContext interface.

--

use **PropertyPlaceholderConfigurer** to share some constant variables to all other beans. For example, define your log file location in a properties file, and access the properties value from different beans configuration files via ${log.filepath}.

--

Spring initializer class that helps us to load Dispatcher Servlet and other Spring configurations.

--

public class WebMvcInitializer extends AbstractAnnotationConfigDispatcherServletInitializer {…}

executed by a special Servlet-ContextInitializer available in the Spring distribution, configures DispatcherServlet and its WebApplicationContext using given metadata sources.

**WebAppInitializer.java**

Spring provides a mechanism in order to create ApplicationContext(ServletContext) without touching web.xml

WebApplicationInitializer is implemented in order to configure the ServletContext programmatically in replacement of the WEB-INF/web.xml file.