**Q1. What is dispatcher-servlet in spring? Why is it used for? Shortly explain about how spring-boot lifecycle works**

**Answer:**

**DispatcherServlet:** DispatcherServlet acts as front controller for Spring based web applications. It provides a mechanism which handles all the HTTP requests and responses and it queues for forwarding to the different controller and methods. It is inherited from javax.servlet.http.HttpServlet.

**DispatcherServlet is used for the following things:**

1. Receives all requests as Front Controller and provides a single entry point to the application
2. Mapping requests to correct Spring MVC controller
3. Consulting ViewResolvers to find correct View
4. Forwarding request to chosen View for rendering
5. Returning the response to the client
6. Creates web-context to initialize the web-specific beans like controllers, view resolvers, and handler mapping

**Spring Boot lifecycle works:**

The application starts using the "main method" which calls the "run" method. From the run method, the main application context kicks off which searches for the classes annotated with @Configuration and initializes all the declared beans in those configuration classes. Based on the scope of those beans, stores those beans in JVM, specifically in a space inside JVM which is known as IOC container. After the creation of all the beans, automatically configures the dispatcher servlet and registers the default handler mappings, messageConverts, and all other basic things.

**Q2. Write a few differences between “git merge” and “git rebase”. When will you choose git merge and when will you choose git rebase?**

**Answer:**

rebase “reapplies commits on top of another base branch”, whereas merge “joins two or more development histories together”. In other words, the key difference between merge and rebase is that while merge preserves history as it happened, rebase rewrites it.

If the feature branch you are getting changes from is shared with other developers, rebasing is not recommended, because the rebasing process will create inconsistent repositories. For individuals, rebasing makes a lot of sense.

If you want to see the history completely same as it happened, you should use merge. Merge preserves history whereas rebase rewrites it.

Rebasing is better to streamline a complex history, you are able to change the commit history by interactive rebase. You can remove undesired commits, squash two or more commits into one or edit the commit message.

Rebase will present conflicts one commit at a time whereas merge will present them all at once. It is better and much easier to handle the conflicts but you shouldn’t forget that reverting a rebase is much more difficult than reverting a merge if there are many conflicts. You can find details of a basic rebase process from git basic rebase.

**Q3. Suppose you made a wrong commit and pushed the code into your remote branch. Now you feel the push needs to be reverted / undoed or the previous head should be back into the remote branch. What would you like to do?**

**Answer:**

This can be done in two ways:-

1. Correct the mistake in a new commit
2. Revert the full commit

**Correct the mistake in a new commit:**

This is simple as I just need to remove or fix the bad file in a new commit and push it to the remote repository. This is the most natural way to fix an error, always safe and totally non-destructive, and how I should do it 99% of the time. Yet the bad commit remains there and accessible, but this is usually not a big deal.

**Revert the full commit:**

In this case, instead of going through all the changes manually I whole previous commit will be reverted. Reverting a commit means to create a new commit that undoes all changes that were made in the bad commit. Just like above, the bad commit remains there, but it no longer affects the current master and any future commits on top of it. To do this, first reset the branch to the parent of the current commit, then force-push it to the remote as follows.

git reset HEAD^ --hard

git push -f

**Q4. What do you understand about J2EE? What are the main J2EE components? Briefly explain EJB.**

**Answer:**

**J2EE** stands for Java 2 Platform Enterprise Edition. J2EE is a platform-independent, Java-centric environment from Sun for developing, building and deploying Web-based enterprise applications online. The J2EE platform consists of a set of services, APIs, and protocols that provide the functionality for developing multi-tiered, Web-based applications.

**Components**: The main components are:

* JavaServer Pages (JSP),
* Java servlets,
* Enterprise JavaBeans (EJB),
* and Java Database Connectivity (JDBC)

**EJB:** Enterprise Java Beans (EJB) is one of the several Java APIs for standard manufacture of enterprise software. EJB is a server-side software element that summarizes business logic of an application. Enterprise Java Beans web repository yields a runtime domain for web related software elements including computer reliability, Java Servlet Lifecycle (JSL) management, transaction procedure and other web services. To run EJB application we need an application server (EJB Container) such as Jboss, Glassfish, Weblogic, Websphere etc. It performs:

1. Life cycle management
2. Security
3. Transaction management
4. Object pooling

**Q5.** **Suppose you started a pizza restaurant with two main types, margarita and classic pizzas. Once customers start coming in, they demand add-ons like mushrooms, onions etc. To save the day, you created subclasses for mushroom and onion. But shortly after, a competitor opens a new restaurant nearby with subclasses for corn, olives, etc. Now considering new competitors, a number of subclasses you will need to create for an effective billing system can go overboard. Which design pattern do you think will be fit most to solve this problem any why? Please briefly explain.**

**Answer:**

I think the Decorator design pattern will be fit most to solve this problem.