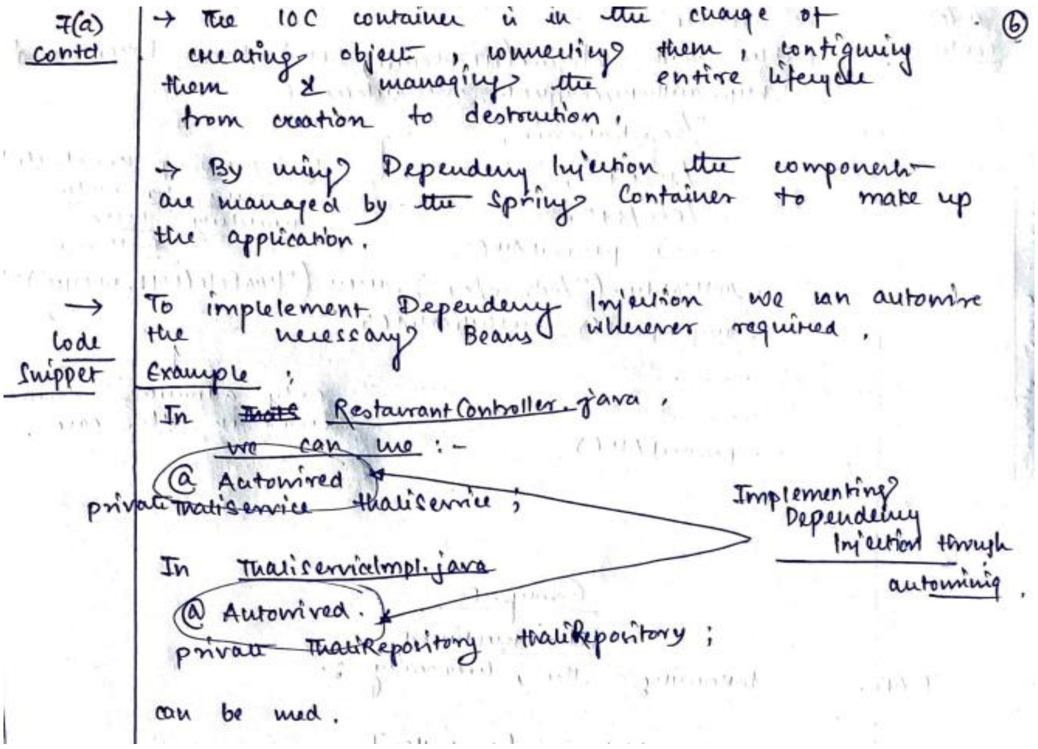
Q..A restaurant deploys a web application for taking online orders………. using the spring framework.

(a) How can dependency injection be implemented in this application?

Ans:



7(a) Contd

→ The IOC container is in the charge of creating object, connecting them, configuring them & managing the entire lifecycle from creation to destruction.

→ By using Dependency Injection the components are managed by the Spring Container to make up the application.

(6)

→ Code Snippet

To implement Dependency Injection we can autowire the necessary Beans wherever required.

Example:

In ~~thats~~ RestaurantController.java, ~~we can use~~ :-

```
@Autowired
private ThaliService thaliService;
```

In ThaliServiceImpl.java

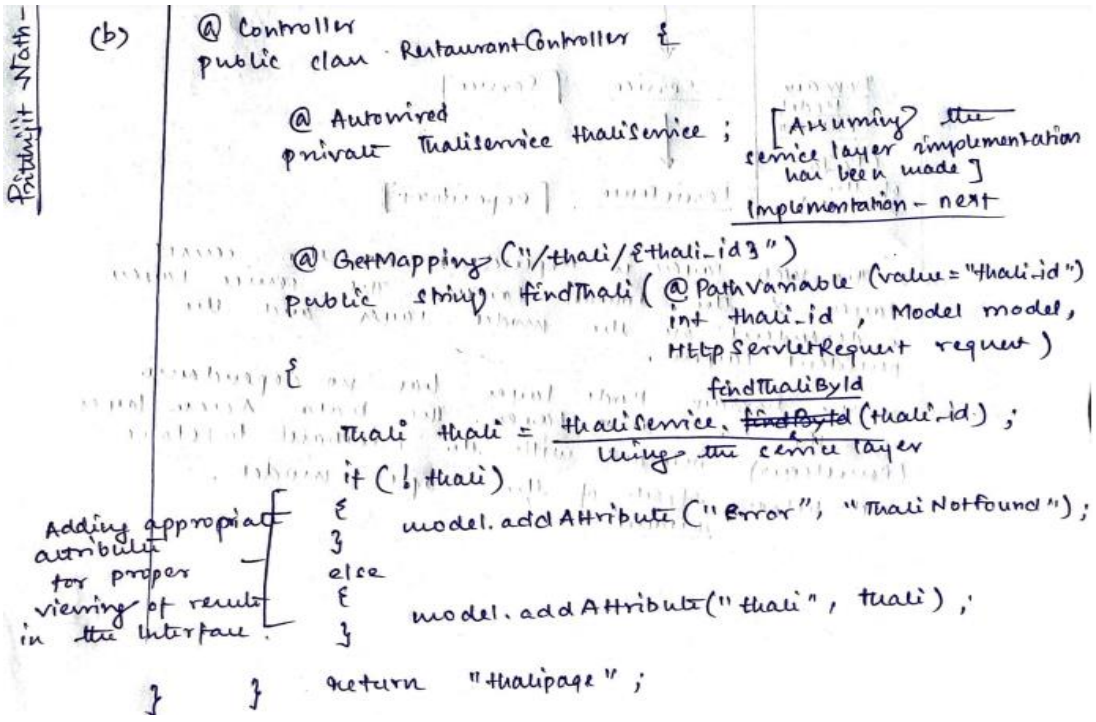
```
@Autowired.
private ThaliRepository thaliRepository;
```

can be used.

Implementing Dependency Injection through autowiring.

(b) If a customer requests for a particular ‘Thali’ (platter), write suitable controller function to check if that is available or not. Let’s assume that you have a java bean named ‘Thali’ and corresponding database entry for that. Justify.

Ans:



Pritthijit Nath

(b)

```
@Controller
public class RestaurantController {

    @Autowired
    private ThaliService thaliService;

    @GetMapping("/thali/{thali-id}")
    public String findThali(@PathVariable(value="thali-id") int thali_id, Model model,
                            HttpServletRequest request)
    {
                            findThaliById
        Thali thali = thaliService.~~findById~~(thali_id);
        if (!thali)
        {
            model.addAttribute("Error", "Thali NotFound");
        }
        else
        {
            model.addAttribute("thali", thali);
        }
        return "thalipage";
}   }
```

[Assuming the service layer implementation has been made]

Implementation - next

using the service layer

Adding appropriate attribute for proper viewing of result in the interface.

```
@ Service                                                        (+)
public class ThaliServImpl implements ThaliService
                                      ~~this is a JPA~~ ↑ ~~interface~~
{                                          ~~<Thali, Long>~~

  ...

    @override Thali findThaliById (int id)
                                          this is JPA interface
    {                                         <Thali, Long>
        Optional<Thali> optional = thaliRepository.
                                   findById (id);
        Thali thali = null;
        if (optional.isPresent()) {
          thali= optional.get()
        }
    }   return thali;
}   ...
```

C). Briefly explain the significance of '@SpringBootApplication' in the context of the mentioned application.

(C)

We use the @SpringBoot Application annotation in our application or main class to enable a host of features. e.g. Java-based Spring Configuration, component scanning, and in particular for enabling Spring Boot's auto configuration feature.

Here, it enables @Configuration, @componentScan and @EnableAutoConfiguaration.

Ans:

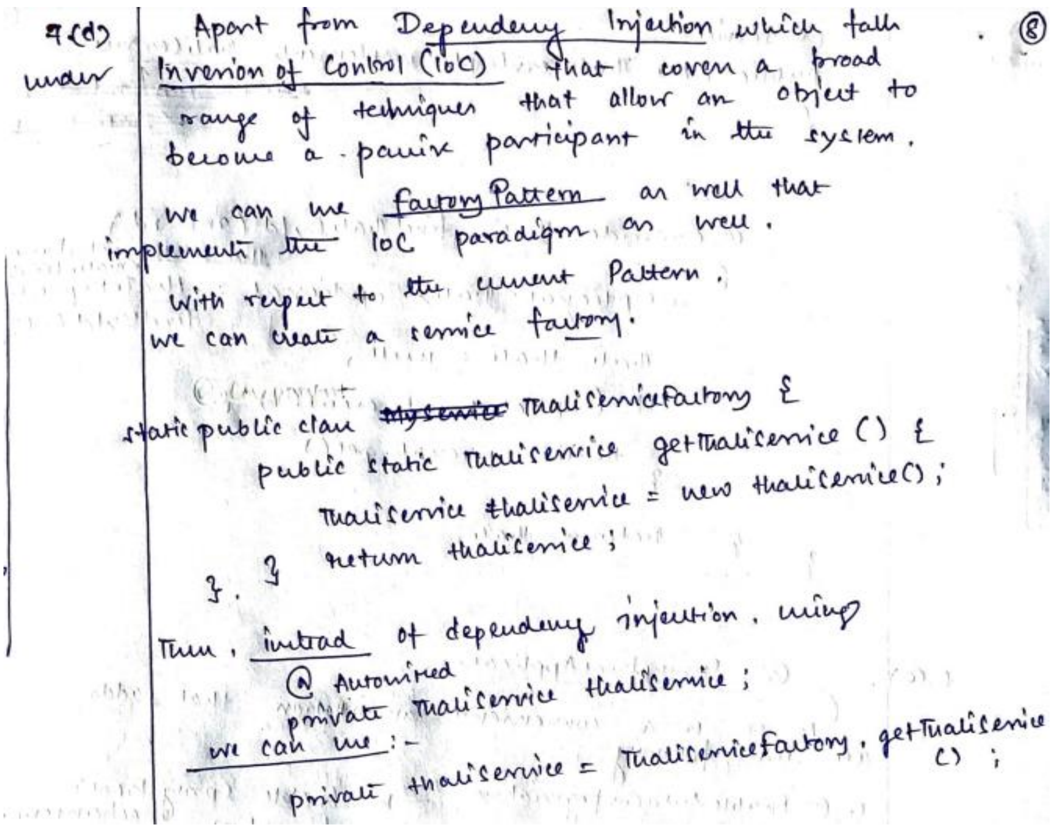
1. @Configuration: this annotation indicates that a class ~~declares~~ one or more @Bean methods and may be processed by spring container to generate bean defination and ~~sp~~ service requests for those beans at runtime.

2. @ComponentScan: This annotation enables component scanning so that the web controller classes and other compoment created will be automatically discovered and registered as beans in spring's Application context. The 'order Platter Controller' and ~~pd~~ 'platterRepository' are found by spring by enabling this annotation.

3. @EnableAutoConfiguration: This annotation eables spring Boot to auto-configure the application context. Therefore, it automatically creates and registers

beans based on both the included jar files in the classpath and the beans defined by us.

D). Discuss about a design pattern apart from dependency injection utilized by Spring. Show how could it be realized w.r.t the current application.



4(d) Apart from Dependency Injection which falls under Inversion of Control (IoC) that covers a broad range of techniques that allow an object to become a passive participant in the system.

⑧

We can use Factory Pattern as well that implements the IoC paradigm as well.

With respect to the current Pattern,
we can create a service factory.

```
static public class ~~MyService~~ ThaliServiceFactory {
        public static ThaliService getThaliService () {
            ThaliService thaliService = new thaliService();
        }  return thaliService;
    }.
```

Thus, instead of dependency injection, using

```
    @Autowired
    private ThaliService thaliService;
```

we can use :-

```
    private thaliService = ThaliServiceFactory.getThaliService
                                                      () ;
```

E). Write the role of object relational mapping in extracting a platter information from the database. Let us assume that a suitable database table exists.

Ans: Object-relational mapping (ORM) is used in Spring to map Java objects to database tables. In the context of extracting platter information from a database, we can use an ORM tool like Hibernate to map the `Thali` Java bean to a corresponding database table. We can use Hibernate's `@Entity` annotation to mark the `Thali` class as an entity, and `@Id` annotation to mark the primary key. We can also use Hibernate's `@Table` annotation to specify the table name and other table-related attributes.

F). How does Spring Boot identify the dependency during application startup?

Spring Boot identifies dependencies during application startup by scanning the application classpath and looking for classes annotated with `@Component`, `@Repository`, `@Service`, `@Controller`, etc. Spring Boot then creates and wires these components to each other automatically.

G). Discuss automatic data marshalling w.r.t Spring framework. Give suitable code snippets.

Ans: Automatic data marshalling is the process of converting Java objects to and from a format that can be stored in a database or transmitted over a network. In Spring, we can use the `@RequestBody` and `@ResponseBody` annotations to automatically marshal and unmarshal data between Java objects and JSON or XML representations.

For example, we can define a REST endpoint that returns a list of available platters in JSON format as follows:

```
@RestController

@RequestMapping("/platters")

public class PlatterController {

  private final PlatterService platterService;



  @Autowired

  public PlatterController(PlatterService platterService) {

      this.platterService = platterService;

  }



  @GetMapping

  public List<Thali> getAvailablePlatters() {

      return
```