# EXPERIMENT NO. 11: DEPENDENCY INJECTION IN SPRING BOOT

## Objective

To create a simple Spring Boot application demonstrating dependency injection.

## Theory

Dependency Injection (DI) is a design pattern that allows for the decoupling of object creation from object usage. In Spring, DI is a core feature that lets components declare their dependencies, which the framework then automatically injects. This promotes modularity, testability, and ease of maintenance.

## Step 1: Create a Spring Boot Project

Use Spring Initializr or an IDE like IntelliJ to create a Spring Boot project with Spring Web dependency.

## Step 2: Create a Service Interface

```
public interface MessageService {

   String getMessage();
}
```

## Step 3: Create a Service Implementation

```
@Service
public class MessageServiceImpl implements MessageService {
   @Override
   public String getMessage() {
      return "Hello, world!";
   }
}
```

## Step 4: Create a Controller

```
@RestController

public class MessageController {
    private final MessageService messageService;

    @Autowired
    public MessageController(MessageService messageService) {
        this.messageService = messageService;
    }

    @GetMapping("/message")
    public String getMessage() {
        return messageService.getMessage();
    }
}
```

## Step 5: Run the Application

Run the Spring Boot application. Open your browser or Postman and navigate to `http://localhost:8080/message`. You should see the response: `Hello, world!`.

## Explanation

In this example, `MessageServiceImpl` implements the `MessageService` interface. The `MessageController` class uses constructor injection (via `@Autowired`) to receive the implementation from Spring. This is a clear example of dependency injection: the controller does not create the service instance, but receives it from the Spring context.

## Conclusion

This experiment demonstrates how Spring Boot uses dependency injection to manage object creation and wiring. Using interfaces and services with dependency injection helps to create flexible, testable, and loosely coupled applications.