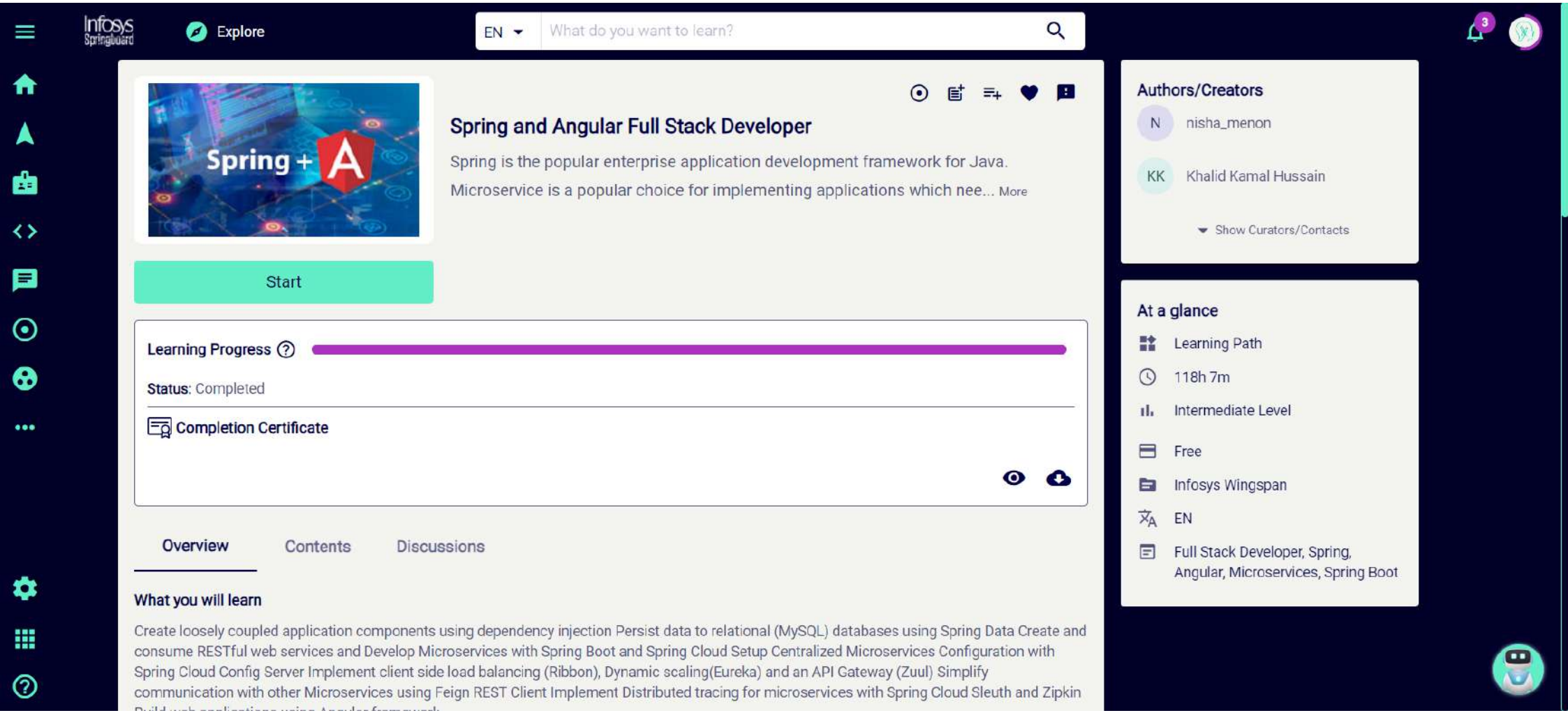

Thank you. Your test is submitted successfully.

You have cleared this assessment.

Obtained Percentage	Obtained Marks
73.33 %	11 / 15

Best Attempt Score:73.33 % on 15-05-2024

Review Your Attempt

Result: CLEARED

Overall Percentage: 73.33% (11/15)

SpringEssentialsAssessment

Section Score: 11/15

Q1 of 15

Consider the following statements regarding bean scope:

1. Bean scope defines the lifetime and visibility of a bean within its container.
2. If a bean is declared with singleton scope, only one instance of that bean will exist in the container.
3. With singleton scope, multiple declarations of the same bean result in all instances pointing to the same object.
4. When a bean is set to prototype scope, each declaration of that bean results in a new instance being created.

Which combination of statements is correct?

- I, II, IV
- I, II, III, IV
- I, III, V
- I, II, V

Q2 of 15

Consider a scenario,where a class has a large number of optional dependencies and needs to maintain flexibility in adding or removing them dynamically without modifying the class code.

Identify from the following which dependency injection method would be most appropriate to use in this situation?

- Field Injection
- Constructor Injection
- Method Injection
- Setter Injection

Q3 of 15

Observe the following classes and interface, assuming all the necessary imports are done.

```
@Service
public class UserService
{
        private EmailService emailService;
        public void sendWelcomeEmail(User user)
        {
        emailService.sendEmail(user.getEmail(), "Welcome to our application!");
        }
}

interface EmailService { void sendEmail(String to, String body); }

@Component
class SmtpEmailService implements EmailService
{
        public void sendEmail(String to, String body)
        {
        // Implementation to send email via SMTP
        }
}



@Component
class MockEmailService implements EmailService
{
        public void sendEmail(String to, String body)
        {
        // Mock implementation for testing
        }
}
```

In the above code, which annotation should be used to inject the SmtpEmailService implementation into the UserService class?

- @Qualifier @Autowired(“smtpEmailService”)
- @Qualifier @Resource(“smtpEmailService”)
- @Autowired @Inject(“smtpEmailService”)
- @Autowired @Qualifier(“smtpEmailService”)

Q4 of 15

Consider the scenario where you are configuring logging for a Spring application and need to set up logging at the ERROR level, log messages into a file and specify the logging pattern to use in that file. Which properties are commonly used for the above?

- logging.level.=ERROR, logging.file.name, logging.pattern.file
- logging.level= ERROR,log.file.path, log.pattern.file
- logging.level.=ERROR,log.file.name , log.pattern.file
- log.level= ERROR ,logging.file.path,log.pattern.file

Q5 of 15

Observe the below code snippet.

```
public class AccountManager
{
        private AccountService accountService;
        public AccountManager(AccountService accountService)
        { this.accountService = accountService; }
        public void performOperation()
        {
                //performs some operations
                accountService.updateBalance();
        }
}

public interface AccountService { void updateBalance(); }
```

Observe the below tester class:

```
public class AccountManagerTester{
        public void testPerformOperation()
        {
                //injecting the mock object
                AccountManager accountManager = new AccountManager(mockAccountService);
                doNothing().when(mockAccountService).updateBalance();
                //perform the operation
                accountManager.performOperation();
                //verify that the updateBalance method of AccountService was called.
                verify(mockAccountservice, times(1)).updateBalance();
        }
}
```

From the below options given, identify the different lines of code which allows you to create the mock object for AccountService interface.

- mockAccountService = Mockito.mock(new AccountService());
- mockAccountService = Mockito.mock(AccountService.class);
- mockAccountService = new AccountService().mock();
- mockAccountService = Mockito.mock(AccountService.java);

Q6 of 15

Consider a Spring Boot application setup for logging with default logging properties.

From the below identify the valid logging properties that can be included in the application.properties file to customize the logging behavior, if needed.

1. <Logging.filename>
2. <Logging.file.name>
3. <Logging.pattern.file>
4. <Logging.level.root>
5. <Logging.file.pattern.file>
6. <Logging.file.console.file>

- (●) Option II, III, IV
- ( ) Option IV, V, VI
- ( ) Option I, II, III
- ( ) Option III, IV and VI

Consider the below scenario.

In a Spring application, the UserService class has a dependency on the UserRepository interface to retrieve user data from a database.

The UserRepository interface has a method getUserById that returns a User object based on the provided user ID.

```
public class UserService
{
        private UserRepository userRepository;
        public UserService(UserRepository userRepository)
        {
                this.userRepository = userRepository;
        }
        public User getUserDetails(int userId)
        {
                return userRepository.getUserById(userId);
        }
}

public interface UserRepository
{
        User getUserById(int userId);
}
```

Given the scenario above, which of the following code snippets correctly configures the UserService and UserRepository beans in a Spring configuration class?

```
@Configuration
public class AppConfig
{
        @Bean
        public UserService userService()
        {
                return new UserService(userRepository());
        }
        @Bean
        public UserRepository userRepository()
        {
                return new UserRepositoryImpl();
        }
}
```

Q8 of 15

Consider the following classes and interface, assuming all the necessary imports are done.

```
@Configuration
@ComponentScan(basePackages = "com.infy.beans")
public class AppConfig { }

public interface Message { String getMessage(); }

@Component
@Qualifier("greeting")
public class GreetingMessgae implements Message
{
        public String getMessage()
        { return "Welcome"; }
}



@Component
@Qualifier("farewell")
public class FarewellMessage implements Message
{
        public String getMessage()
        { return "GoodBye"; }
  }



@Component
public class Printer
{
        private Message message;
        @Autowired public Printer(@Qualifier Message message)
        { this.message=message; }

        public void printMessage()
        { System.out.println(message.getMessage()); }
}
```

Predict the output.

Predict the output.

- NullPointerException
- Display “GoodBye” on the console
- Display “Welcome” on the console
- UnsatisfiedDependencyException

Q9 of 15

Consider the following CartService class and its configuration:

```
@Service
public class CartService
{
        private PricingService pricingService;
        @Autowired
        public void setPricingService(PricingService pricingService)
        {
                this.pricingService = pricingService;
        }
        // Other methods...
}



interface PricingService { // Pricing service methods... }

 @Component
 class DefaultPricingService implements PricingService
 {
 // Implementation...
 }
```

In the above code, which of the following statements is correct?

- The DefaultPricingService will not be injected into the CartService because it is not annotated with @Service.
- The DefaultPricingService will be injected into the CartService using the @Autowired annotation on the setter method.
- The DefaultPricingService will not be injected into the CartService because it should be injected via constructor injection.
- A NoSuchBeanDefinitionException will be thrown at runtime because there is no bean definition for PricingService.

Q10 of 15

Consider the following classes and interface, assuming all the necessary imports are done.

```
package com.example.service;
@Component(value = "paymentService")
public class PaymentServiceImpl
{
        private PaymentGateway paymentGateway;
}

public interface PaymentGateway { void processPayment(double amount); }

package com.example.config;
@Configuration @ComponentScan(basePackages = "com.example.service")
public class AppConfig { }
```

Which of the following code snippets correctly obtain an instance of the PaymentServiceImpl bean class from the Spring container? (Select the two correct options)

- ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class); PaymentServiceImpl ps = context.getBean(PaymentServiceImpl.class);
- ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class); PaymentServiceImpl ps = context.getBean("paymentService");
- ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig); PaymentServiceImpl ps = (PaymentServiceImpl) context.getBean("paymentService");
- ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class); PaymentServiceImpl ps = context.getBean("PaymentServiceImpl");

Q11 of 15

Consider the below class CalculatorTester.java which has a test method testAddition() in it.

```
public class CalculatorTester{ @Test void testAddition()
{
// Junit Passes
List list = Mockito.mock(ArrayList.class);
list.add("Spring Data");
list.add("Spring REST");
when(list.get(eq(3))).thenReturn("Spring Essentials");
Assertions.assertEquals("Spring Essentials", list.get(3));     // line 1
Assertions.assertEquals(2, list.size());      // line 2
}
}
```

What will be output when the test method gets executed?

- Compilation error at line 3
- Test method fails due to AssertionFailedError at line 2
- Test method fails due to AssertionFailedError at line 1
- Test method is passed.

Q12 of 15

In a Spring application, which of the following naming patterns is commonly used for profile specific application properties file?

- application.properties
- application_profile.properties
- application-profile.properties
- profile-application.properties

Q13 of 15

Suppose you have a logging aspect in your application , and you want to execute advice before any method in classes located in the package 'com.example.services' with method names starting with "calculate".

Which of the following pointcut expressions correctly specifies this scenario?

1. @Before("execution(* com.example.services.*.calculate*(..))")
2. @Before("execution(* com.example.services.calculate*(..))")
3. @Before("execution(* com.example.services.*.**(..))")
4. @Before("execution(* com.example.services..calculate*(..))")

- (●) Option I , II, III
- ( ) Option II , III
- ( ) Option I
- ( ) Option I, IV

Q14 of 15

From the below options choose all the method's signature that matches with the given pointcut expression:

execution(* com.example.service.*Impl.fetch Customer(..))"))

Options:

1. void com.example.service.EmployeeImpl#fetchCustomer()
2. int com.example.service.EmployeeImpl#fetchCustomer(int a)
3. void com.example.service.EmployeeImpl#fetchCustomer(int a, int b)
4. void com.example.service.Employee#fetchCustomer()

- (●) Options I,II, III
- ( ) Options I and II
- ( ) Options I, II and IV
- ( ) Options I and III

Q15 of 15

Consider the below scenario in a Spring application.Suppose you have a service layer in your application that contains various business logic methods. You want to log the execution time of methods in this service layer using AOP. Which of the following configurations correctly implements this logging functionality in Spring AOP?

```
@Aspect
@Component
public class LoggingAspect {

    private static final Logger LOGGER = LoggerFactory.getLogger(LoggingAspect.class);

    @Around("execution(* com.example.service.*.*(..))")
    public Object logExecutionTime(ProceedingJoinPoint joinPoint) throws Throwable {
    long startTime = System.currentTimeMillis();

    Object result = joinPoint.proceed();

    long endTime = System.currentTimeMillis();
    long executionTime = endTime - startTime;

    String className = joinPoint.getTarget().getClass().getSimpleName();
    String methodName = joinPoint.getSignature().getName();

    LOGGER.info("{}.{}() executed in {} ms", className, methodName, executionTime);

    return result.
}
```