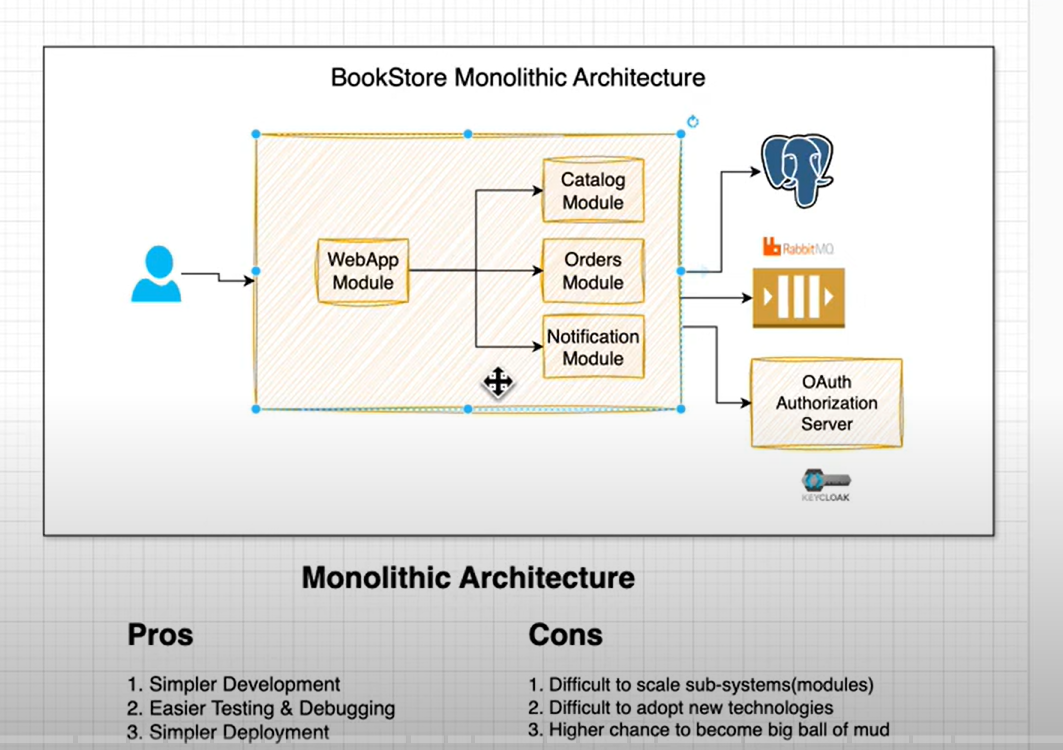
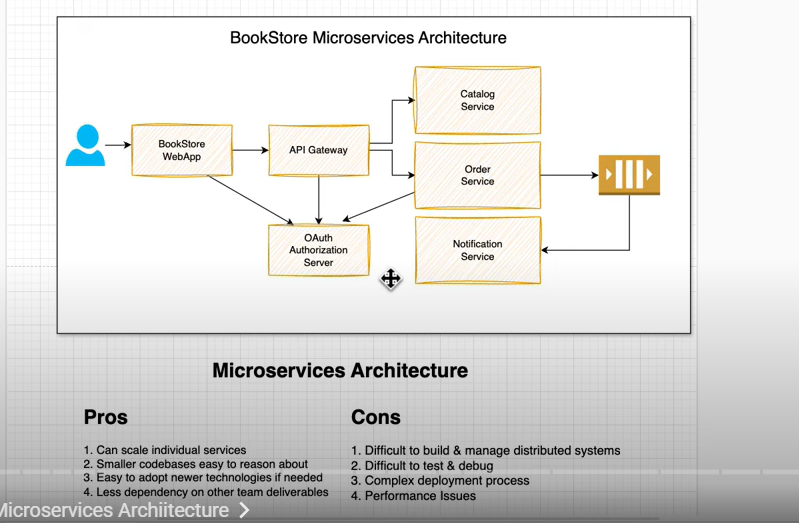
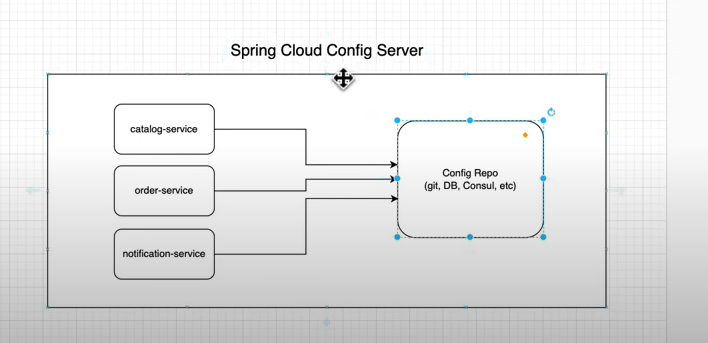
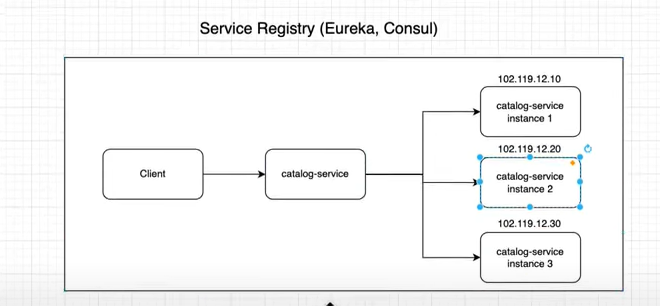
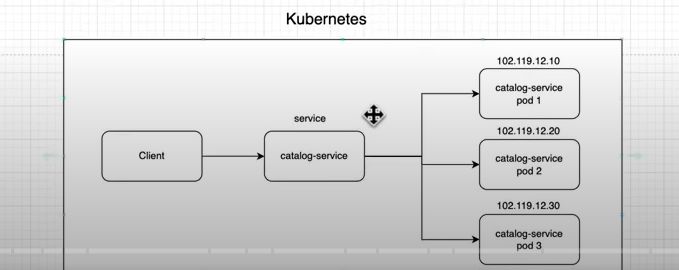
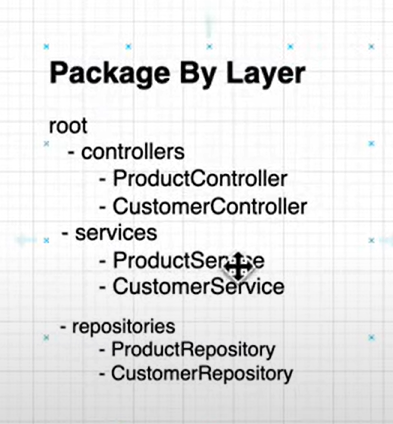
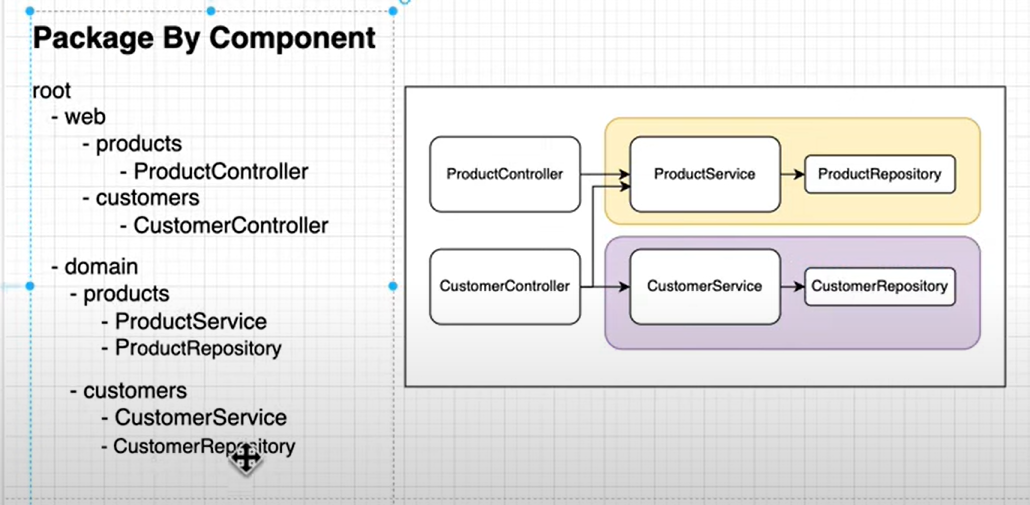
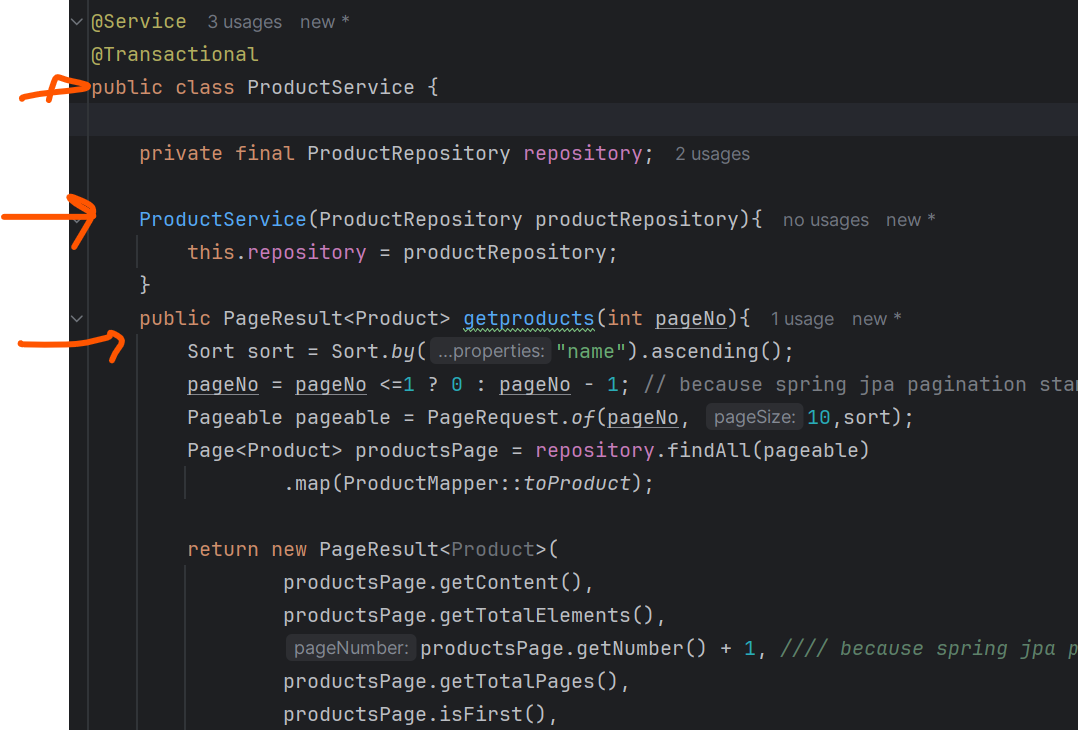
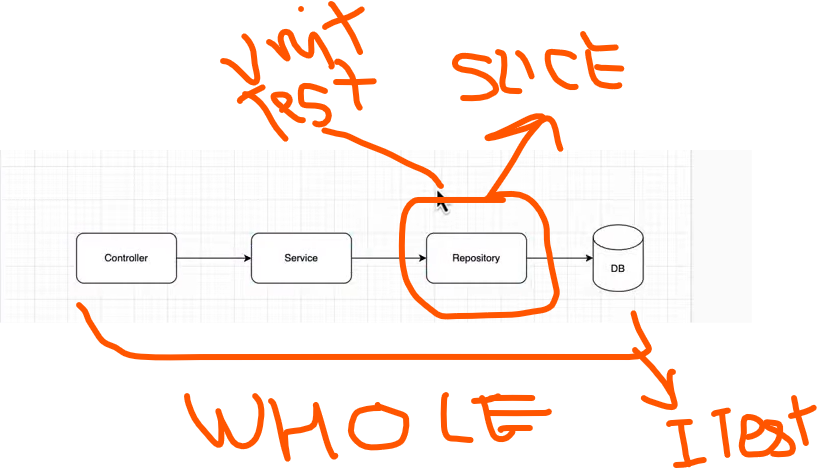
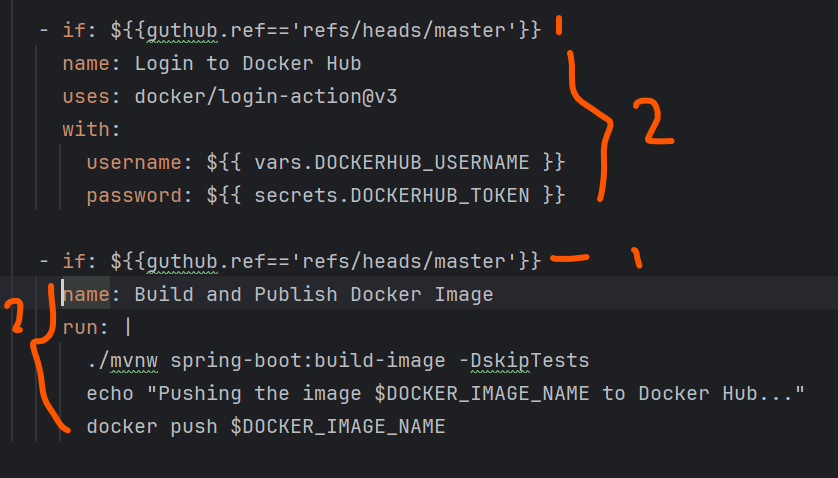
Spring Boot Microservices project:

1. Monolith Architecture:
   1. 
2. Microservices Architecture:
   1. 
3. Parameters to choose tech stack to develop an application.
   1. You can think of choosing the most advanced technology available in the market to develop your application, but it’s not always the right
   2. You also need to think about the team you have the skill set they are good at , you can’t always hire resource train them for new technology.
   3. We need to consider the timelines to deliver the application, based on which we can decide to do experiment on new technologies or go with best people we have who can work and deliver on time.
   4. Consider cost of tech stack which you choose if you have budget constraints as well.
   5. Example: choose Rabbit MQ vs Kafka , which one you choose you your app?
      1. Most of us know that Kafka is rich in features and lot more advanced then RabbitMQ and easily can make decision to pick Kafka over RabbitMQ.
      2. But Being and Architect or tech lead you don’t only think about which one is better rather which one is better as per project requirements. Does Kafka and Rabbit MQ both serves the purpose ? then Does our team has a good understanding of Rabbit MQ? Do we have budget constraints? If Rabbit MQ Serves the purpose and it’s under budget constraints and our team also has good control over Rabbit MQin production env we should definitely go for Rabbit MQ rather Kafka.
4. Spring cloud Config Server:
   1. When we implement microservice we need to do lot of configure , typically in properties or yml files. So, instead of keeping them in specific service it’s a good practice to use a central repository to keep these configs and when we spin up the services they can get the necessary properties in time.
   2. Spring cloud config server is a native spring module which provides this feature.
   3. 
5. Kubernetes:
   1. If we are using k8s we mostly used configMap and secrets to store configuration and secrets.
6. Service registry:
   1. In microservice environment we usually keep multiple instance running for each service, service registry keeps track of all live instances and remove terminated instances so that client can talk to services registry and SR can delegate the call the live instance.
   2. There are multiple solutions available to implement service registry concept.
      1. Eureka , Consul, ETC.
      2. 
   3. Similar concept available in Kubernetes , there is a concept of services in k8s all services register in services.yml file with logical public facing service name instead if directly communicating with pods.
   4. 
7. Best practices for package structure:
   1. Package by layer:
      1. 
   2. Package by Feature:
      1. A white grid with black text

         AI-generated content may be incorrect.
   3. Package by component:
      1. The problem with above 2 approach is that you make your repository public and put validation logics in service layer. Now, anyone can call inject and use your repository and persist invalid data to let say product bypassing service layer logic.
      2. With package by component approach, you can make you repository package – private so that only your service can access it any once can talk to service, service will perform the validation and forward the call to repository post validation.
      3. 
8. Dependencies:
   1. Org.Flyway-db
9. **Important points:**
   1. Any internal utility class do not have to be public , they should be package private.
   2. **Controller class** : controllers do not have to be public because spring framework going to invoke this , no body else can invoke it.
   3. All the handlers method in controller do not have to be public too for same reason.
   4. Eg:
      1. 
   5. **Service Class:**
      1. Service class has to be public because we need to call it from controller.
      2. Methods inside it which we call has to be public,
      3. no need to make constructors as public.
      4. Eg: 
   6. **Repository Service:**
      1. Repository does not have to be public it should be package private so that only intended service can call it no other class can.
   7. **sd**
10. **Unit Test and Integration Test:**
    1. ****
    2. Use base class for common configuration to avoid writing same code again and again.
    3. **Unit test** :
       1. **UT** are written only for single components like controller, service or repository.
       2. **Repository**:
          1. **@DataJpaTest** annotation is used to test JPA repository.
          2. Also called slice test annotation because we are only testing splice of our application.
          3. This annotation is going to spin up in memory data base to connect and interact with it.
       3. Service:
    4. **Integration tests** :
       1. **IT** are written for whole application combined. It loads the entire application context to test end-2-end.
       2. Common annotations:
          1. @SpringBootTest
          2. @TestConfiguration
          3. @Sql("/test-data.sql"):
             1. During test execution it is always recommended to use confirmed data rather predicted data. Means , let say we have **test1**: which get all records from db and assert all conditions and we have one script which insert that data into db so far everything looks fine. But, when we have another **test2 in** same class which insert another product into table then if test2 runs first test 1 will fail with it’s condition.
             2. @SQL annotation ensure that it will run before every test . so even if any other test deleted or inserted data into table we can create script to truncate the data and insert again for specific test case.
       3. Always keep on testing changes after doing small amount if changes to avoid issues.
11. Docker :
    1. Usually to build docker image we need to create dockerFile and specify all the steps to build docker image.
    2. But, with spring-boot-maven-plugin spring makes it very easy to create docker images without even using docker file. behind the scenes these plugin uses cloud native buildpacks and build docker image
    3. **Generate docker hub access token:**
       1. dockerHub Account > account setting> setting > personal access token> generate token
       2. A screenshot of a computer

          AI-generated content may be incorrect.
       3. Token value: dckr\_pat\_3PvHbWQCpeTBvzcf8Pg5HO4vga0
    4. **Configure docker hub cred as secrets in Github.**
       1. Login to github repositories
       2. Goto secrets and variables tab > actions > repository secrets.
       3. repository secrets. : create/save dockerhub\_username and dockerhub\_token.
       4. A screenshot of a computer

          AI-generated content may be incorrect.
       5. After saving these details we can use github actions called login-action to login into dockerhub account
       6. Login-action url: [docker/login-action: GitHub Action to login against a Docker registry](https://github.com/docker/login-action)
       7. Below screenshot shown login to github , build docker image and only when code pushed to master branch.
          1. 
       8. d
    5. fsd
12. Sd
13. Sds
14. Important command:
    1. Goto project from ubunutu:
       1. cd /mnt/c/Users/hemkumar11/intellijWorkspace/sivalabs/BookStoreWorkspace/spring-boot-microservices-course
    2. build docker image:
       1. ./mvnw -pl catalog-service spring-boot:build-image -DskipTests
       2. -pl catalog-service : indicate build only catalog service if command runs on root module.
    3. sd
15. END