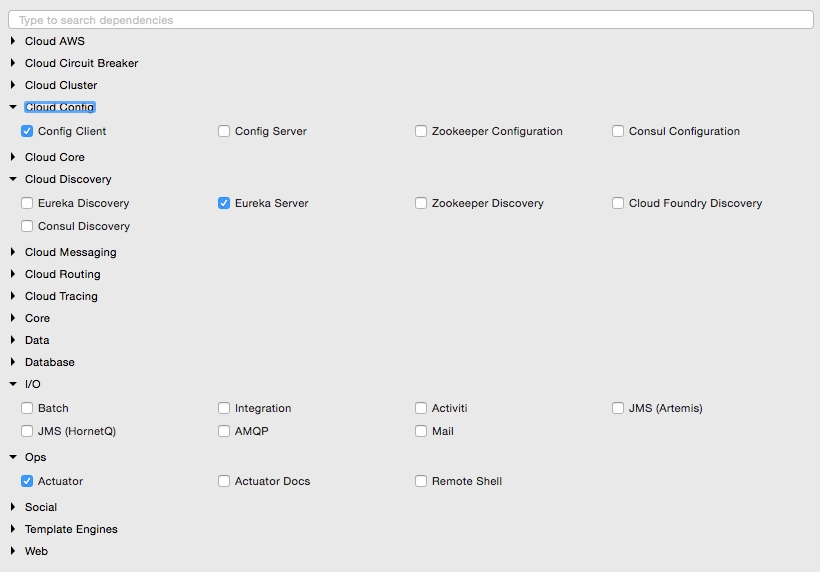
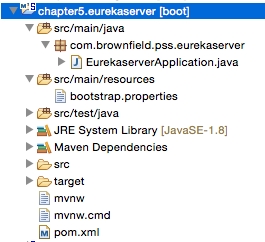
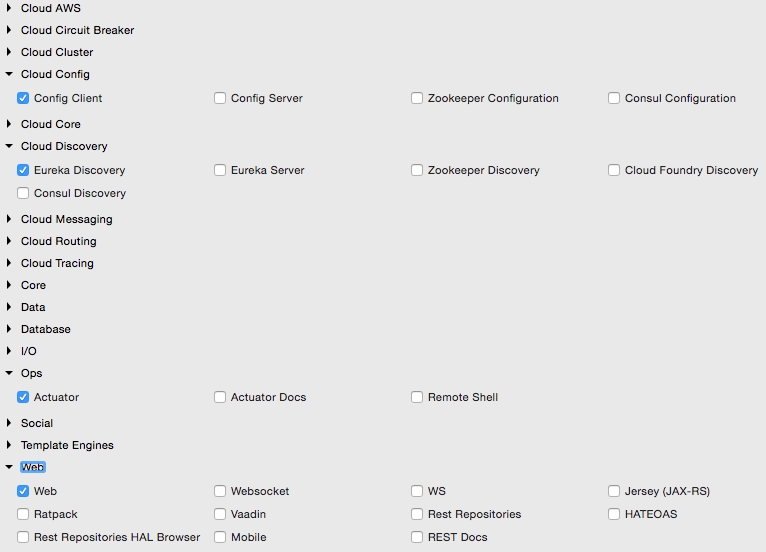
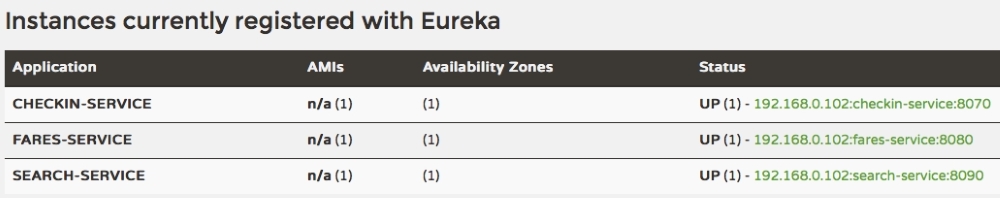
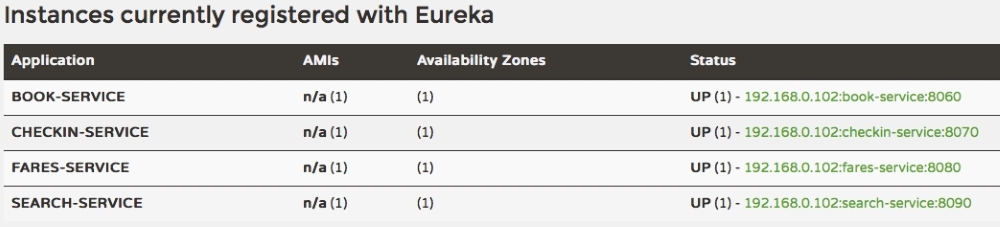
* Lab 15 - Setting up the Eureka server
* Start a new Spring Starter project, and select **Config Client**, **Eureka Server**, and **Actuator**:
* 
* The project structure of the Eureka server is shown in the following image:
* 
* Note that the main application is named EurekaserverApplication.java.
* Rename application.properties to bootstrap.properties since this is using the Config server. As we did earlier, configure the details of the Config server in the bootsratp.properties file so that it can locate the Config server instance. The bootstrap.properties file will look as follows:
* spring.application.name=eureka-server1  
  server.port:8761  
  spring.cloud.config.uri=http://localhost:8888
* The Eureka server can be set up in a standalone mode or in a clustered mode. We will start with the standalone mode. By default, the Eureka server itself is another Eureka client. This is particularly useful when there are multiple Eureka servers running for high availability. The client component is responsible for synchronizing state from the other Eureka servers. The Eureka client is taken to its peers by configuring the eureka.client.serviceUrl.defaultZone property.
* In the standalone mode, we point eureka.client.serviceUrl.defaultZone back to the same standalone instance. Later we will see how we can run Eureka servers in a clustered mode.
* Create a eureka-server1.properties file, and update it in the Git repository. eureka-server1 is the name of the application given in the application's bootstrap.properties file in the previous step. As shown in the following code, serviceUrl points back to the same server. Once the following properties are added, commit the file to the Git repository:
* spring.application.name=eureka-server1  
  eureka.client.serviceUrl.defaultZone:http://localhost:8761/eureka/  
  eureka.client.registerWithEureka:false  
  eureka.client.fetchRegistry:false
* Change the default Application.java. In this example, the package is also renamed as com.brownfield.pss.eurekaserver, and the class name changed to EurekaserverApplication. In EurekaserverApplication, add @EnableEurekaServer:
* @EnableEurekaServer  
  @SpringBootApplication  
  public class EurekaserverApplication {
* We are now ready to start the Eureka server. Ensure that the Config server is also started. Right-click on the application and then choose **Run As** | **Spring Boot App**. Once the application is started, open http://localhost:8761 in a browser to see the Eureka console.
* In the console, note that there is no instance registered under **Instances currently registered with Eureka**. Since no services have been started with the Eureka client enabled, the list is empty at this point.
* Making a few changes to our microservice will enable dynamic registration and discovery using the Eureka service. To do this, first we have to add the Eureka dependencies to the pom.xml file. If the services are being built up fresh using the Spring Starter project, then select **Config Client**, **Actuator**, **Web** as well as **Eureka discovery** client as follows:
* 
* Since we are modifying our microservices, add the following additional dependency to all microservices in their pom.xml files:
* <dependency>  
   <groupId>org.springframework.cloud</groupId>  
   <artifactId>spring-cloud-starter-eureka</artifactId>  
  </dependency>
* The following property has to be added to all microservices in their respective configuration files under config-repo. This will help the microservices to connect to the Eureka server. Commit to Git once updates are completed:
* eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
* Add @EnableDiscoveryClient to all microservices in their respective Spring Boot main classes. This asks Spring Boot to register these services at start up to advertise their availability.
* Start all servers except Booking. Since we are using the Ribbon client on the Booking service, the behavior could be different when we add the Eureka client in the class path. We will fix this soon.
* Going to the Eureka URL (http://localhost:8761), you can see that all three instances are up and running:
* 
* Time to fix the issue with Booking. We will remove our earlier Ribbon client, and use Eureka instead. Eureka internally uses Ribbon for load balancing. Hence, the load balancing behavior will not change.
* Remove the following dependency:
* <dependency>  
   <groupId>org.springframework.cloud</groupId>  
   <artifactId>spring-cloud-starter-ribbon</artifactId>  
  </dependency>
* Also remove the @RibbonClient(name="fares") annotation from the FareServiceProxy class.
* Update @FeignClient(name="fares-service") to match the actual Fare microservices' service ID. In this case, fare-service is the service ID configured in the Fare microservices' bootstrap.properties. This is the name that the Eureka discovery client sends to the Eureka server. The service ID will be used as a key for the services registered in the Eureka server.
* Also remove the list of servers from the booking-service.properties file. With Eureka, we are going to dynamically discover this list from the Eureka server:
* fares-proxy.ribbon.listOfServers=localhost:8080, localhost:8081
* Start the Booking service. You will see that CommandLineRunner successfully created a booking, which involves calling the Fare services using the Eureka discovery mechanism. Go back to the URL to see all the registered services:
* 
* Change the website project's bootstrap.properties file to make use of Eureka rather than connecting directly to the service instances. We will not use the Feign client in this case. Instead, for demonstration purposes, we will use the load balanced RestTemplate. Commit these changes to the Git repository:
* spring.application.name=test-client  
  eureka.client.serviceUrl.defaultZone: http://localhost:8761/eureka/
* Add @EnableDiscoveryClient to the Application class to make the client Eureka-aware.
* Edit both Application.java as well as BrownFieldSiteController.java. Add three RestTemplate instances. This time, we annotate them with @Loadbalanced to ensure that we use the load balancing features using Eureka and Ribbon. RestTemplate cannot be automatically injected. Hence, we have to provide a configuration entry as follows:
* @Configuration  
  class AppConfiguration {  
   @LoadBalanced  
   @Bean  
   RestTemplate restTemplate() {  
   return new RestTemplate();  
   }  
  }  
  @Autowired  
  RestTemplate searchClient;  
     
  @Autowired  
  RestTemplate bookingClient;  
     
  @Autowired  
  RestTemplate checkInClient;
* We use these RestTemplate instances to call the microservices. Replace the hardcoded URLs with service IDs that are registered in the Eureka server. In the following code, we use the service names search-service, book-service, and checkin-service instead of explicit host names and ports:
* Flight[] flights = searchClient.postForObject("http://search-service/search/get", searchQuery, Flight[].class);  
    
  long bookingId = bookingClient.postForObject("http://book-service/booking/create", booking, long.class);  
     
  long checkinId = checkInClient.postForObject("http://checkin-service/checkin/create", checkIn, long.class);
* We are now ready to run the client. Run the website project. If everything is fine, the website project's CommandLineRunner will successfully perform search, booking, and check-in. The same can also be tested using the browser by pointing the browser to http://localhost:8001.