CST-339 Activity 7 Guide

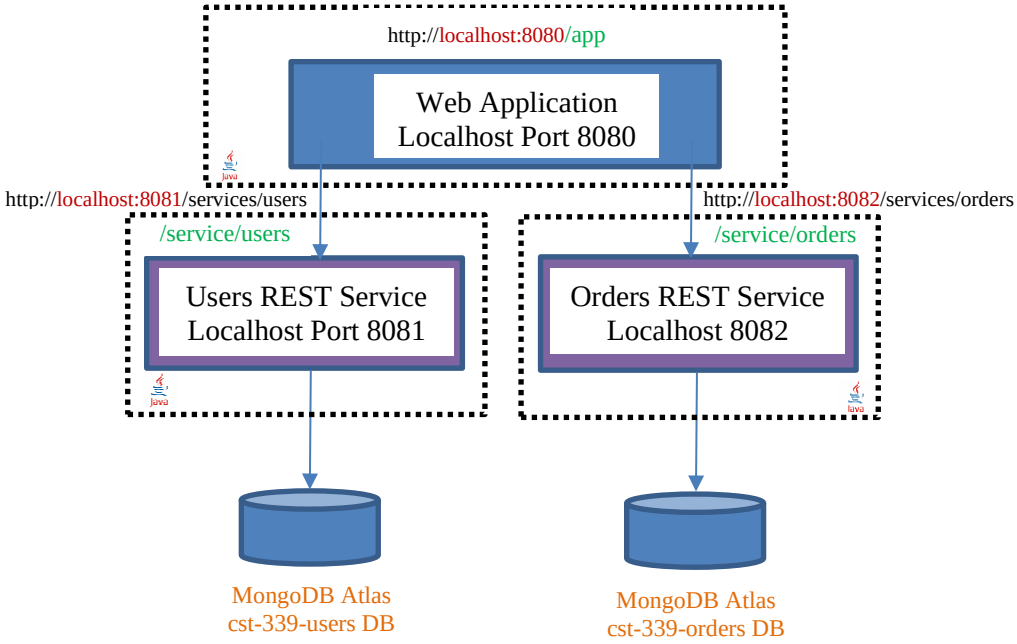
Contents

Part 1: Building a Web Application that Consumes Microservices

Overview

Goal and Directions:

In this activity, you will use build a small web application using Spring MVC that will consume 2 microservices exposed over 2 REST API endpoints, all designed using a microservices architecture. The design will consist of 3 independent Spring Boot applications. The microservices architecture will allow you to build, deploy, and scale each application independently of each other. To keep the design simple, the application and services will not be secured.



Execution

Execute this assignment according to the following guidelines:

1. We will build a couple of simple REST Services and then create a simple web application.
2. To build the Users REST API complete the following steps:
    a. Log into your MongoDB Atlas account. Create a new database named *cst339-user*s. Add a new User Collection named *users* to this database with an id, username, and password properties all of type String. Populate 5–10 test users.
    b. Create a new Spring Boot Starter project named *topic7-1a*. Add the *Spring Web, Spring Data MongoDB*, and *Spring Boot DevTools* dependencies to the project.
    c. Copy the *UsersRepository* class and the *UserEntity* class from the *topic6-3* project into this project.
    d. Create a new class name *UserModel* in the *com.gcu.model* package. Add an id, username, and password properties, all of type String with getter and setter methods. Add a default constructor that will initialize the all the properties to default values. Add a non-default constructor that will initialize all properties from the passed in method arguments.

```
public class UserModel
{
    private String id;
    private String username;
    private String password;

    public UserModel()
    {
        this.id = "";
        this.username = "";
        this.password = "";
    }
    public UserModel(String id, String username, String password)
    {
        super();
        this.id = id;
        this.username = username;
        this.password = password;
    }
```

    e. Add a new business service class named *UserBusinessService* in the *com.gcu.business* package. The class should be marked with the *@Service* annotation. Autowire the *UsersRepository* into variable named *service* using constructor injection. Add a new method *public List<UserModel> getAllUsers()* that gets all users from the database using the *findAll*() method on the *usersRepository* variable. Convert the list of *UserEntity* to a list of *UserModel* and return the list of *UserModel*.

```java
@Service
public class UserBusinessService
{
    @Autowired
    private UsersRepository usersRepository;

    /**
     * Non-Default constructor for constructor injection.
     */
    public UserBusinessService(UsersRepository usersRepository)
    {
        this.usersRepository = usersRepository;
    }

    /**
     * Get all Users from the database.
     */
    public List<UserModel> getAllUsers()
    {
        // Get all all the Entity Users
        List<UserEntity> usersEntity = usersRepository.findAll();
        // Iterate over the Entity Users and create a list of Domain Users
        List<UserModel> usersDomain = new ArrayList<UserModel>();
        for(UserEntity user : usersEntity)
        {
            usersDomain.add(new UserModel(user.getId(), user.getUsername(), user.getPassword()));
        }

        // Return list of Domain Users
        return usersDomain;
    }
}
```

f. Add a new REST service class named *UsersRestService* to the project in the *com.gcu.api* package. The class should be marked with the *@RestController* annotation and *@RequestMapping("/service")* annotation. Add an API named *getUsers()* that is marked with the *@GetMapping("/users")* annotation that takes no method arguments. Inject an instance of the *UserBusinessService* named *service* into the class. The implementation should call the *getAllUsers()* method from the *service* variable. The method should return an instance of *ResponseEntity* with a response code of HttpStatus.OK if no error and a response code of HttpStatus.INTERNAL_SERVER_ERROR in a try catch block.

```java
@RestController
@RequestMapping("/service")
public class UsersRestService
{
    @Autowired
    UserBusinessService service;

    @GetMapping(path="/users")
    public ResponseEntity<?> getUsers()
    {
        try
        {
            List<UserModel> users = service.getAllUsers();
            if(users == null)
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            else
                return new ResponseEntity<>(users, HttpStatus.OK);
        }
        catch (Exception e)
        {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

g. Configure the application setting the port for the application to 8081 and the MongoDB Atlas database to cst-339-users by adding the following properties to the *application.properties* file.

```
# Application Port
server.port = 8081

# MongoDB Atlas Database
spring.data.mongodb.uri=mongodb+srv://mark:gcu-bssd@cluster0-ihiam.mongodb.net/test?retryWrites=true&w=majority
spring.data.mongodb.database=cst339-users
```

h. Test the REST API in your browser by navigating to the *http://localhost:8081/service/users* URL.

3. To build the Orders REST API complete the following steps:
    a. Log into your MongoDB Atlas account. Create a new database named *cst339-order*s. Add a new Order Collection named *orders* to this database with id, orderNo, and productName of type String, price of type Decimal128, and quantity of type Int32. Populate 5–10 test orders.
    b. Create a new Spring Boot Starter project named *topic7-1b*. Add the *Spring Web, Spring Data MongoDB*, and *Spring Boot DevTools* dependencies to the project.
    c. Copy the *OrdersRepository* class, the *OrderModel* class, and the *OrderEntity* class from the *topic6-3* project into this project. Add a default constructor to the *OrderModel* class that will initialize the all the properties to default values. Add a non-default constructor that will initialize all properties from the passed in method arguments.

```
public class OrderModel
{
    @Id
    String id = "";
    String orderNo = "";
    String productName = "";
    float price = 0;
    int quantity = 0;

    public OrderModel()
    {
        this.id = "";
        this.orderNo = "";
        this.productName = "";
        this.price = 0;
        this.quantity = 0;
    }
    public OrderModel(String id, String orderNo, String productName, float price, int quantity)
    {
        this.id = id;
        this.orderNo = orderNo;
        this.productName = productName;
        this.price = price;
        this.quantity = quantity;
    }
```

d. Add a new business service class named *OrdersBusinessService* in the *com.gcu.business* package. The class should be marked with the *@Service* annotation. Autowire the *OrdersRepository* into variable named *service* using constructor injection. Add a new method *public List<OrderModel> getAllOrders()* that gets all orders from the database using the *findAll*() method on the *ordersRepository* variable. Convert the list of *OrderEntity* to a list of *OrderModel* and return the list of *OrderModel*.

```java
@Service
public class OrdersBusinessService
{
    @Autowired
    private OrdersRepository ordersRepository;

    /**
     * Non-Default constructor for constructor injection.
     */
    public OrdersBusinessService(OrdersRepository ordersRepository)
    {
        this.ordersRepository = ordersRepository;
    }

    /**
     * Get all the Orders from the database.
     */
    public List<OrderModel> getAllOrders()
    {
        // Get all all the Entity Orders
        List<OrderEntity> ordersEntity = ordersRepository.findAll();

        // Option 2: Iterate over the Entity Orders and create a list of Domain Orders
        List<OrderModel> ordersDomain = new ArrayList<OrderModel>();
        for(OrderEntity entity : ordersEntity)
        {
            ordersDomain.add(new OrderModel(entity.getId(), entity.getOrderNo(), entity.getProductName(), entity.getPrice(), entity.getQuantity()));
        }

        // Return list of Domain Orders
        return ordersDomain;
    }
}
```

e. Add a new REST service class named *OrdersRestService* to the project in the *com.gcu.api* package. The class should be marked with the *@RestController* annotation and *@RequestMapping("/service")* annotation. Add an API named *getOrders()* that is marked with the *@GetMapping("/orders")* annotation that takes no method arguments. Inject an instance of the *OrdersBusinessService* named *service* into the class. The implementation should call the *getAllOrders()* method from the *service* variable. The method should return an instance of *ResponseEntity* with a response code of HttpStatus.OK if no errors and a response code of HttpStatus.INTERNAL_SERVER_ERROR in a try catch block.

```
@RestController
@RequestMapping("/service")
public class OrdersRestService
{
    @Autowired
    OrdersBusinessService service;

    @GetMapping(path="/orders")
    public ResponseEntity<?> getOrders()
    {
        try
        {
            List<OrderModel> orders = service.getAllOrders();
            if(orders == null)
                return new ResponseEntity<>(HttpStatus.NOT_FOUND);
            else
                return new ResponseEntity<>(orders, HttpStatus.OK);
        }
        catch (Exception e)
        {
            return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
        }
    }
}
```

f. Configure the application setting the port for the application to 8082 and the MongoDB Atlas database to cst-339-orders by adding the following properties to the *application.properties* file.

```
# Application Port
server.port = 8082

# MongoDB Atlas Database
spring.data.mongodb.uri=mongodb+srv://mark:gcu-bssd@cluster0-ihiam.mongodb.net/test?retryWrites=true&w=majority
spring.data.mongodb.database=cst339-orders
```

g. Test the REST API in your browser by navigating to the *http://localhost:8082/service/orders* URL.

4. To build the Web Application complete the following steps:
   a. Create a new Spring Boot Starter project named *topic7-1c*. Add the *Spring Web, Thymeleaf,* and *Spring Boot DevTools* dependencies to the project. Update the POM file to also include the *thymeleaf-layout-dialect* library.
   b. Copy the Thymeleaf layouts and static image content *topic6-3* project into this project. Remove the Log Out button from the NavBar.
   c. Copy the OrderModel and the UserModel classes from the *topic7-1a* and *topic7-1b* projects.
   d. Create a new file *TestController.java* in the *com.gcu.controller* package. Add a *@RequestMapping("/app")* annotation to the class to route all requests to the */app* URI. Add a Controller Request method named *home(Model model)* to handle a GET request to the root / URI that displays a view named *home*. Create the *home* view that simply has an anchor tag to the */getusers* controller route and another anchor tag to the */getorders* controller route.

```
@Controller
@RequestMapping("/app")
public class TestController
{
    @GetMapping("/")
    public String home(Model model)
    {
        // Display the Home page
        model.addAttribute("title", "Demo Microservices Application");
        return "home";
    }
```

e. Add a Controller Request method named *getUsers(Model model)* to handle a GET request to the */getusers* URI that calls the */service/users* REST API running at localhost on port 8081 using an instance of a *RestTemplate* class, which takes the returned *List<UserModel>* and passes this to a view named *users*. Create the *users* view that simply displays the list of Users in an HTML table.

```
@GetMapping("/getusers")
public String getUsers(Model model)
{
    // Create the REST API end point URL
    String hostname = "localhost";
    int port = 8081;

    // Get all the Users from the REST API
    String url = "http://" + hostname + ":" + port + "/service/users";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<UserModel>> rateResponse = restTemplate.exchange(url, HttpMethod.GET, null, new ParameterizedTypeReference<List<UserModel>>() {});
    List<UserModel> users = rateResponse.getBody();

    // Display the Users
    model.addAttribute("title", "List of Users");
    model.addAttribute("users", users);
    return "users";
}
```

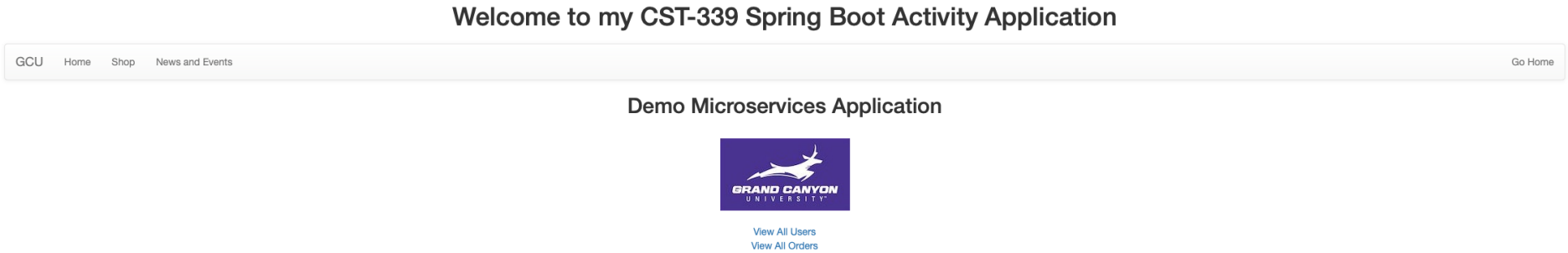
f. Add a Controller Request method named *getOrders(Model model)* to handle a GET request to the */getorders* URI that calls the */service/orders* REST API running at localhost on port 8082 using an instance of a *RestTemplate* class, which takes the returned *List<OrderModel>* and passes this to a view named *orders*. Create the *orders* view that simply displays the list of Orders in an HTML table.

```java
@GetMapping("/getorders")
public String getOrders(Model model)
{
    // Create the REST API end point URL
    String hostname = "localhost";
    int port = 8082;

    // Get all the Orders from the REST API
    String url = "http://" + hostname + ":" + port + "/service/orders";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<OrderModel>> rateResponse = restTemplate.exchange(url, HttpMethod.GET, null, new ParameterizedTypeReference<List<OrderModel>>() {});
    List<OrderModel> orders = rateResponse.getBody();

    // Display the Orders
    model.addAttribute("title", "List of Orders");
    model.addAttribute("orders", orders);
    return "orders";
}
```

g. Start the Spring Boot applications for each of the Users and Orders REST API's if they are not running. Test the Web application in your browser by navigating to the *http://localhost:8080/app/* URL and testing that both the users and orders can be properly displayed. Take a screenshot of the users displayed in an HTML table. Take a screenshot of the orders displayed in an HTML table. Stop all applications.



Welcome to my CST-339 Spring Boot Activity Application

GCU Home Shop News and Events Go Home

List of Users

User Name	Password
Mark	Reha
Mary	Reha
Justine	Reha
Brianna	Reha

Go Home

Welcome to my CST-339 Spring Boot Activity Application

GCU Home Shop News and Events Go Home

List of Orders

Order Number	Product Name	Price	Quantity
0000000001	Product 1	1.0	1
0000000002	Product 2	1.0	1
0000000003	Product 3	5.0	10
0000000004	Product 4	10.0	100

Go Home

Deliverables:

The following needs to be submitted as this part of the activity:

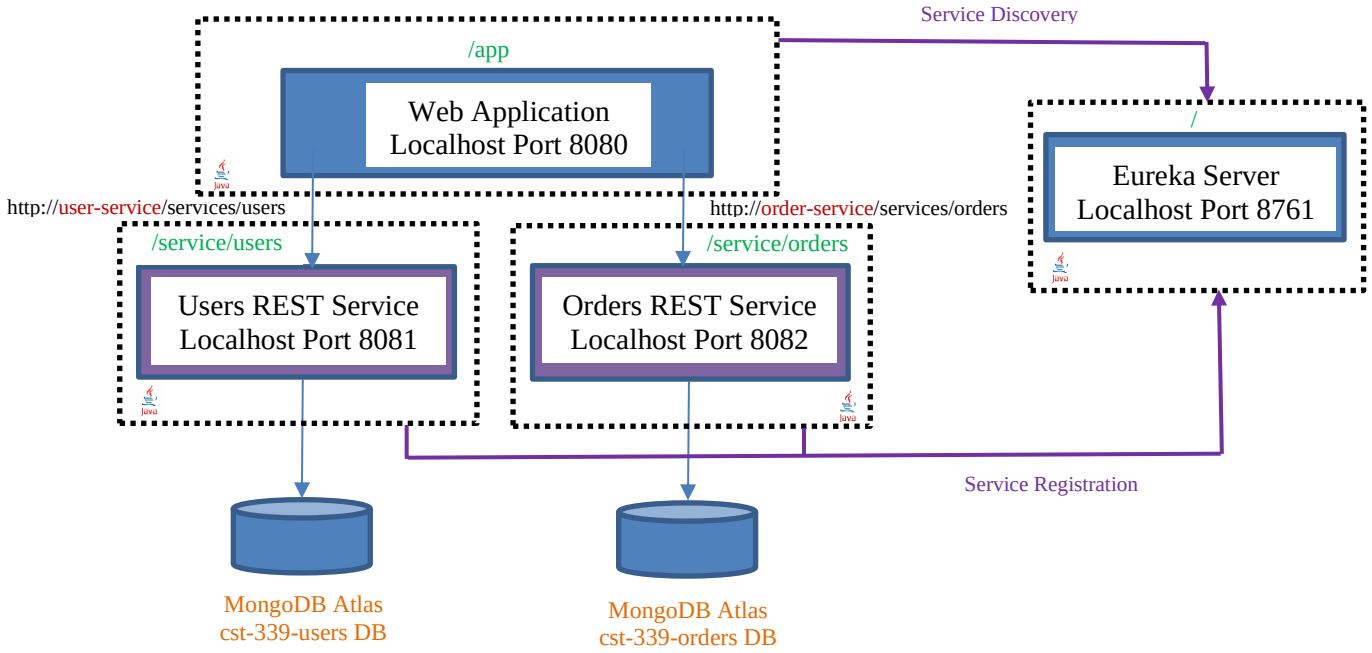
a. Screenshot of the users displayed in a table.
b. Screenshot of the orders displayed in a table.

Part 2: Integrating a REST Service Registry and Discovery Service

Overview

Goal and Directions:

In this activity, you will improve the microservices application built in Part 1 by integrating a REST Service Registry and Discovery Service. This will allow the web application, which is consuming the 2 REST API's, to remove the "hardwired" endpoint hostname and port from the application, allowing the REST API's to be relocated without impacting any applications that are consuming the REST API's. This activity will use the Netflix Eureka Server that is available from Spring Boot libraries.



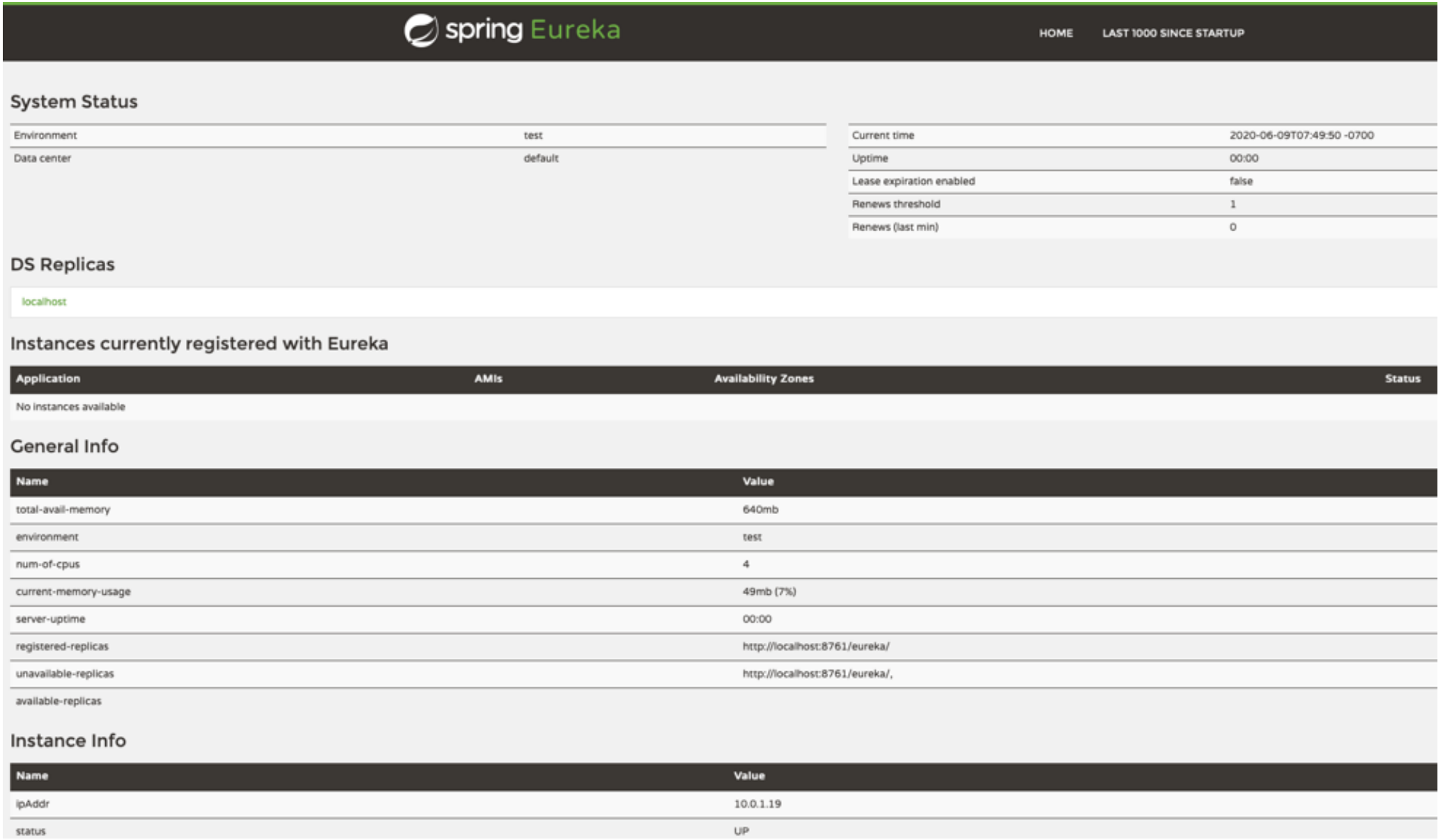
Execution

Execute this assignment according to the following guidelines:

1. We will build an application to run the Eureka Server and then modify the 3 applications from Part 1 to integrate with the Eureka Server.
2. To build the Eureka Server complete the following steps:
    a. Create a new Spring Boot Starter project named *topic7-2es*. Add the *Spring Eureka Server* and *Spring Boot Actuator* dependencies.
    b. Add the *@EnableEurekaServer* annotation to the main application class.
    c. Create a new file in the *src/main/resources* directory named *application.yml* with the following Eureka Server configuration.

```
server:
  port: 8761
eureka:
  client:
    registerWithEureka: false
    fetchRegistry: false
```

    d. Test the Web application in your browser by navigating to the *http://localhost:8761/* URL. The default Eureka Server dashboard should be displayed with no Applications registered. Take a screenshot of the Eureka Server dashboard.

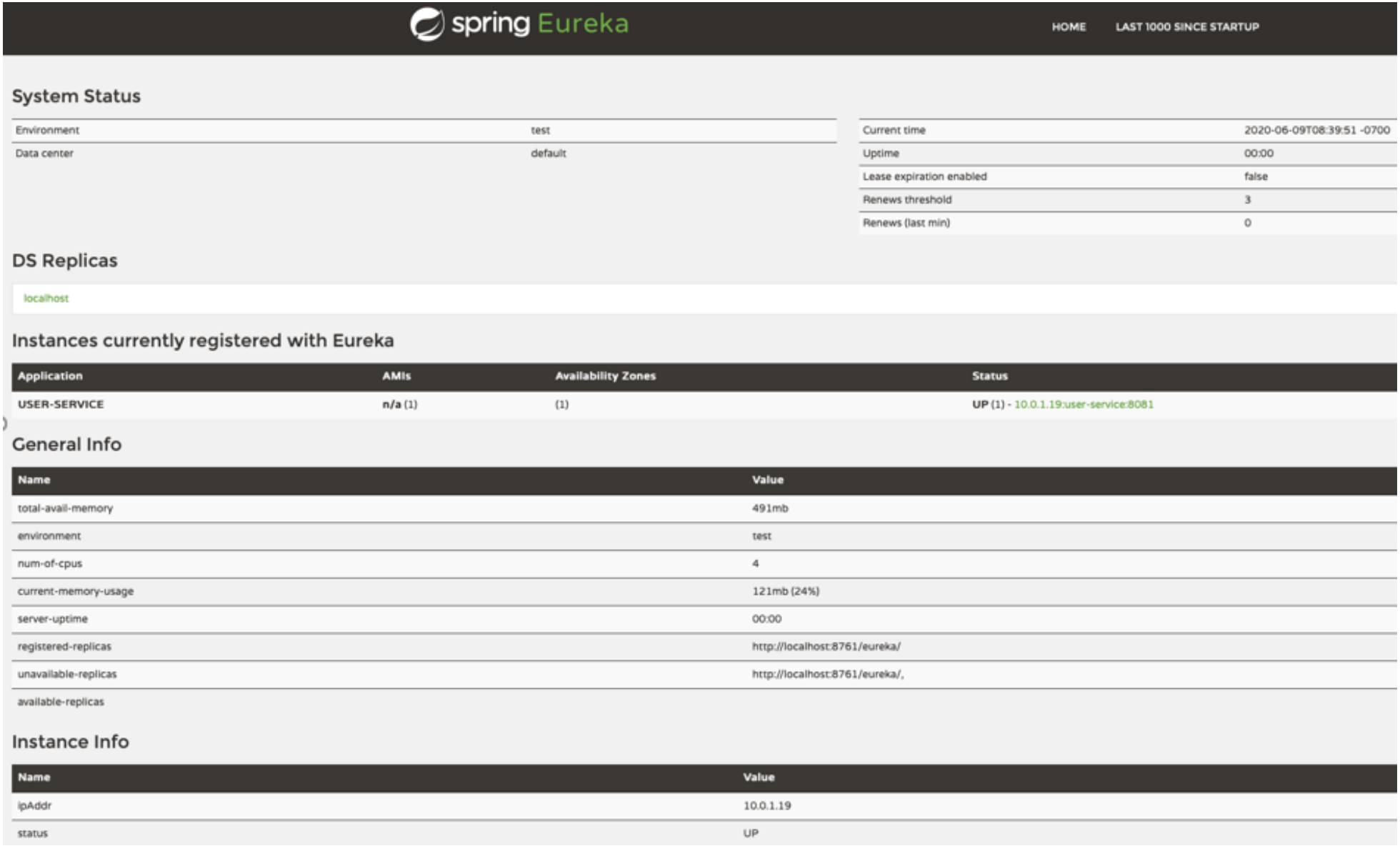


3. To update the Users and Orders REST API's so they are registered with the Eureka Server, complete the following steps:
    a. Make a copy of the *topic7-1a* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic7-2a*. Rename the Spring application file from *Topic71Application.java* to *Topic72Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic7-2a*. Run the project to ensure that everything is working properly.
    b. Add the *Spring Cloud Dependencies*, *Eureka Server*, and *Spring Boot Actuator* dependencies to the POM file (see the entries from the Eureka Server POM file). Make sure to add the *spring-cloud.version* property in the POM file.
    c. Add the *@EnableEurekaClient* annotation to the main application class.
    d. Create a new file in the *src/main/resources* directory named *application.yml* with the following Eureka Server configuration. Note the Spring application name.

```yaml
spring:
  application:
    name: user-service   #current service name to be used by the eureka server

eureka:         #tells about the Eureka server details and its refresh time
  instance:
    leaseRenewalIntervalInSeconds: 1
    leaseExpirationDurationInSeconds: 2
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    healthcheck:
      enabled: true
```

    e. Start the REST API application. The Eureka Server dashboard should now be updated to display a new registered application named *USER-SERVICE*. Take a screenshot of the Eureka Server dashboard.



f. Make a copy of the *topic7-1b* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic7-2b*. Rename the Spring application file from *Topic71Application.java* to *Topic72Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic7-2b*. Run the project to ensure that everything is working properly

g. Add the *Spring Cloud Dependencies*, *Eureka Server*, and *Spring Boot Actuator* dependencies to the POM file (see the entries from the Eureka Server POM file). Make sure to add the *spring-cloud.version* property in the POM file.

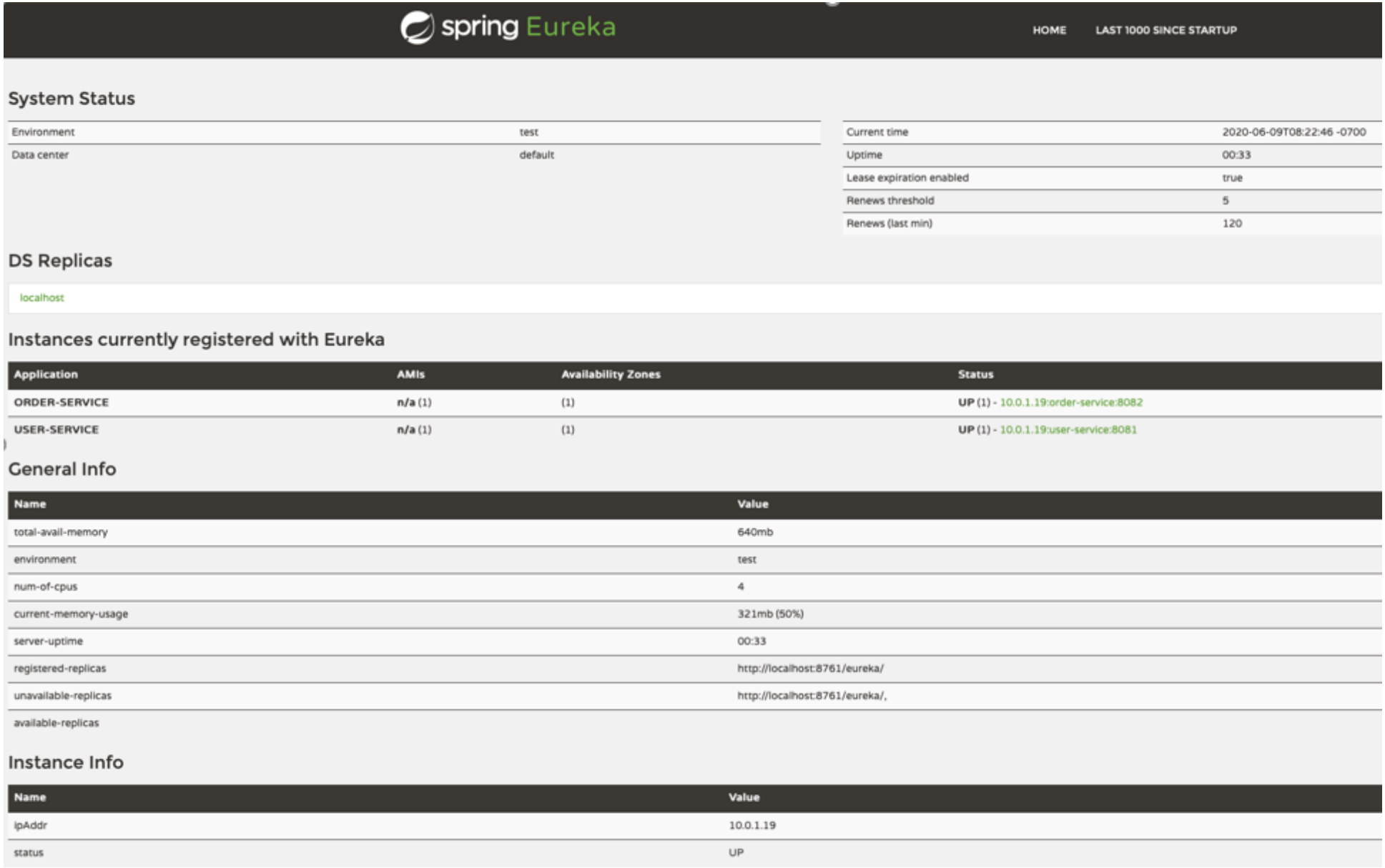
h. Add the *@EnableEurekaClient* annotation to the main application class.

i. Create a new file in the *src/main/resources* directory named *application.yml* with the following Eureka Server configuration. Note the Spring application name.

```yaml
spring:
  application:
    name: order-service   #current service name to be used by the eureka server

eureka:          #tells about the Eureka server details and its refresh time
  instance:
    leaseRenewalIntervalInSeconds: 1
    leaseExpirationDurationInSeconds: 2
  client:
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    healthcheck:
      enabled: true
```

j. Start the REST API application. The Eureka Server dashboard should now be updated to display a new registered application named *ORDER-SERVICE*. Take a screenshot of the Eureka Server dashboard.



4. To update the web application so it discovers the RESPI API's registered with the Eureka Server, complete the following steps:
    a. Make a copy of the *topic7-1c* project by right-clicking on the project, selecting the Copy menu option, then right-clicking in the Workspace, and then selecting the Paste menu option. Name the project *topic7-2c*. Rename the Spring application file from *Topic71Application.java* to *Topic72Application.java*. Fix up the *artifactId* and *name* tags in the POM file (pom.xml) so the value is set to *topic7-2c*. Run the project to ensure that everything is working properly.
    b. Add the *Spring Cloud Dependencies*, *Eureka Discovery Client*, and *Spring Boot Actuator* dependencies to the POM file (see the entries from the Eureka Server POM file). Make sure to add the *spring-cloud.version* property in the POM file.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
```

    c. Create a new file in the *src/main/resources* directory named *application.yml* with the following Eureka Server configuration.

```
eureka:         #tells about the Eureka server deta
  instance:
    leaseRenewalIntervalInSeconds: 1
    leaseExpirationDurationInSeconds: 2
  client:
    registerWithEureka: false
    fetchRegistry: true
    serviceUrl:
      defaultZone: http://localhost:8761/eureka/
    healthcheck:
      enabled: true
```

    d. Update the *TestController* class to look up the REST API endpoint hostname and port by completing the following steps:
        i. Inject an instance variable named *eurekaClient* of type *EurekaClient* as a class-scoped variable.
        ii. Update the *getUsers()* method by using the *eurekaClient* variable to look up the *user-service* (i.e., Eureka application) by calling the *getApplication()* method, get the first instance of the application into local variable *instanceInfo* of type *InstanceInfo*, and then get the hostname and port from the *instanceInfo* variable by calling the *getHostName()* method and *getPort()* method. Create the complete endpoint URL of the Users REST API by using the hostname and port and using this URL in the Rest template.

```java
@GetMapping("/getusers")
public String getUsers(Model model)
{
    // Look up the Host Name and Port for the Users REST API
    Application application =  eurekaClient.getApplication("user-service");
    InstanceInfo instanceInfo = application.getInstances().get(0);
    String hostname = instanceInfo.getHostName();
    int port = instanceInfo.getPort();

    // Get all the Users from the REST API
    String url = "http://" + hostname + ":" + port + "/service/users";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<UserModel>> rateResponse = restTemplate.exchange(url, HttpMethod.GET, null, new ParameterizedTypeReference<List<UserModel>>() {});
    List<UserModel> users = rateResponse.getBody();

    // Display the Users
    model.addAttribute("title", "List of Users");
    model.addAttribute("users", users);
    return "users";
}
```

iii. Update the *getOrders()* method by using the *eurekaClient* variable to look up the *order-service* (i.e., Eureka application) by calling the *getApplication()* method, get the first instance of the application into local variable *instanceInfo* of type *InstanceInfo*, and then get the hostname and port from the *instanceInfo* variable by calling the *getHostName()* method and *getPort()* method. Create the complete endpoint URL of the Orders REST API by using the hostname and port and using this URL in the Rest template.

```java
@GetMapping("/getorders")
public String getOrders(Model model)
{
    // Look up the Host Name and Port for the Users REST API
    Application application =  eurekaClient.getApplication("order-service");
    InstanceInfo instanceInfo = application.getInstances().get(0);
    String hostname = instanceInfo.getHostName();
    int port = instanceInfo.getPort();

    // Get all the Orders from the REST API
    String url = "http://" + hostname + ":" + port + "/service/orders";
    RestTemplate restTemplate = new RestTemplate();
    ResponseEntity<List<OrderModel>> rateResponse = restTemplate.exchange(url, HttpMethod.GET, null, new ParameterizedTypeReference<List<OrderModel>>() {});
    List<OrderModel> orders = rateResponse.getBody();

    // Display the Orders
    model.addAttribute("title", "List of Orders");
    model.addAttribute("orders", orders);
    return "orders";
}
```

e. Start the Eureka Server if it is not running. Start the Spring Boot applications for each of the Users and Orders REST API's if they are not running. Validate that both REST API's are registered in the Eureka Server. Test the web application in your browser by navigating to the *http://localhost:8080/app/* URL and testing that both the users and orders can be properly displayed. Take a screenshot of the users displayed in an HTML table. Take a screenshot of the orders displayed in an HTML table. Stop all applications.

Deliverables:

The following needs to be submitted as this part of the activity:

a. Screenshots of the Eureka Server dashboard with both REST API's registered.
b. Screenshot of the users displayed in a table.
c. Screenshot of the orders displayed in a table.

Research Questions

1. Research Questions: For traditional ground students, answer the following questions in a Microsoft Word document:
    a. Research microservices. Describe what they are. How does this architecture style differ from traditional monolithic architectures?
    b. Research microservices. What are 5 challenges you might encounter when modifying a monolithic architecture to this architecture style?
    c. Christian worldview: Let's say you are asked to build a new application using all the frameworks we studied in the course. The application must help solve a problem that improves the environment in some way. For examples and ideas, Google "improving the environment." Provide a list of 10 User Stories in UML format that support the requirements to solve your environmental problem. Also provide a UML component and deployment diagram of your application, clearly outlining the components, purpose, and how you could showcase the technologies learned in this course to help you solve the problem for the common good.

Final Activity Submission

1. In a Microsoft Word document, complete the following for the Activity Report:
    a. Cover Sheet with the name of this assignment, date, and your name.
    b. Section with a title that contains all the screenshots for each part of the activity.
    c. Section with a title that contains the answers to the research questions (traditional ground students only).
2. Submit the Activity Report as directed by your instructor.