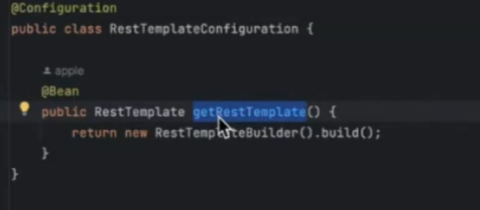
* For a dependency we can inject it using either Autowired annotation or constructor injection.
* RestTemplate impl is already a part of spring web
* We create beans through configuration classes.

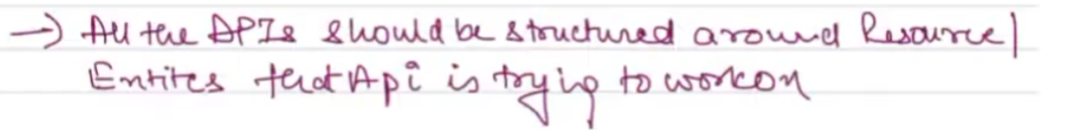
@Bean -> This means whatever obj is getting returned from a func is registered as a Bean.

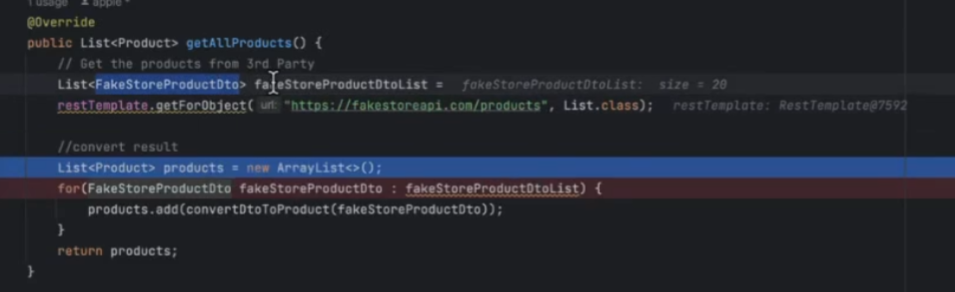


Like we are creating the obj and reg is as a bean using @Bean

BeanObject -> This is an object managed by Spring no matter how it got created. For eg if I put annotation @Service then in this case spring is creating an object. In this case spring creates a BEAN.

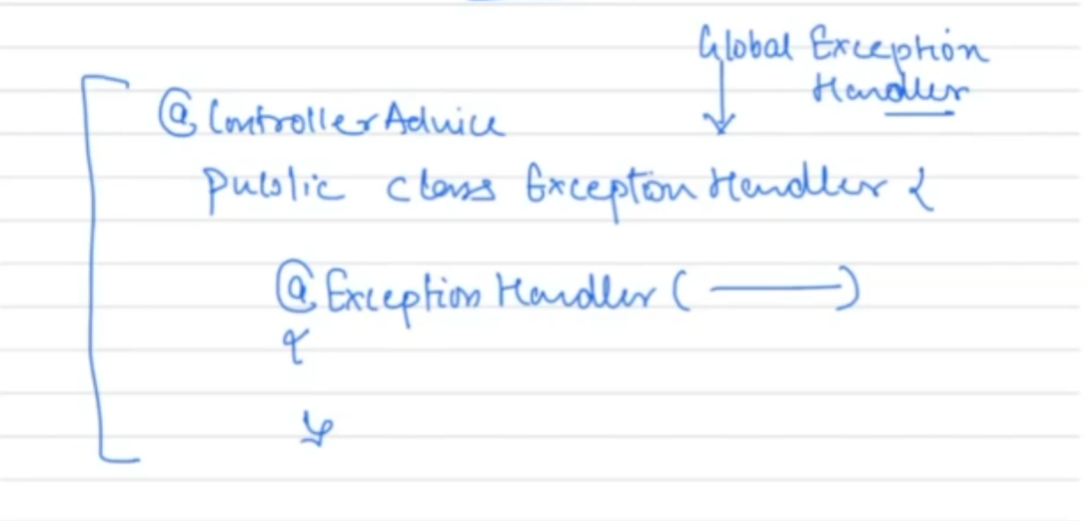
  
Now here in controller we do not have @Bean but we are using @RestController so Spring will internalling create and obj and reg is as a Bean.

* 
* 

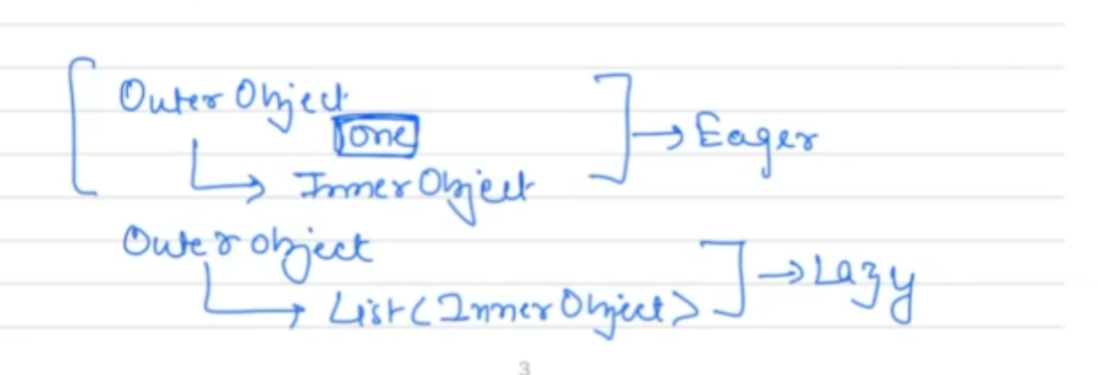
With this kind of impl for the getForObject call we are sending List.class.   
Now in this case it is returning us a result of of list with 20 items but the type is LinkedHashMap. Now this will throw a run time exception as we try to iterate over this list.  
  
  
  
Also when java makes a call for the list of results it ignore the type check we have put. This is done to support backward compatibility and gets a result as LinkedHashMap, although we are collecting the result in List as FakeStoreProductDto. This is known as **Type Erasure.**There are 2 solns to handle this:

1. In the for loop instead of FakeStoreProductDto use LinkedHaspMap and then write a convertor.
2. Avoid usage of List.class and use below format

* @ControllerAdvice helps us to set up a global exception handler instead of handling each exception individually at the Controller level. This is a common piece of code so its more modular. Also this ensure SRP as controller should not be in charge of handling and throwing exceptions

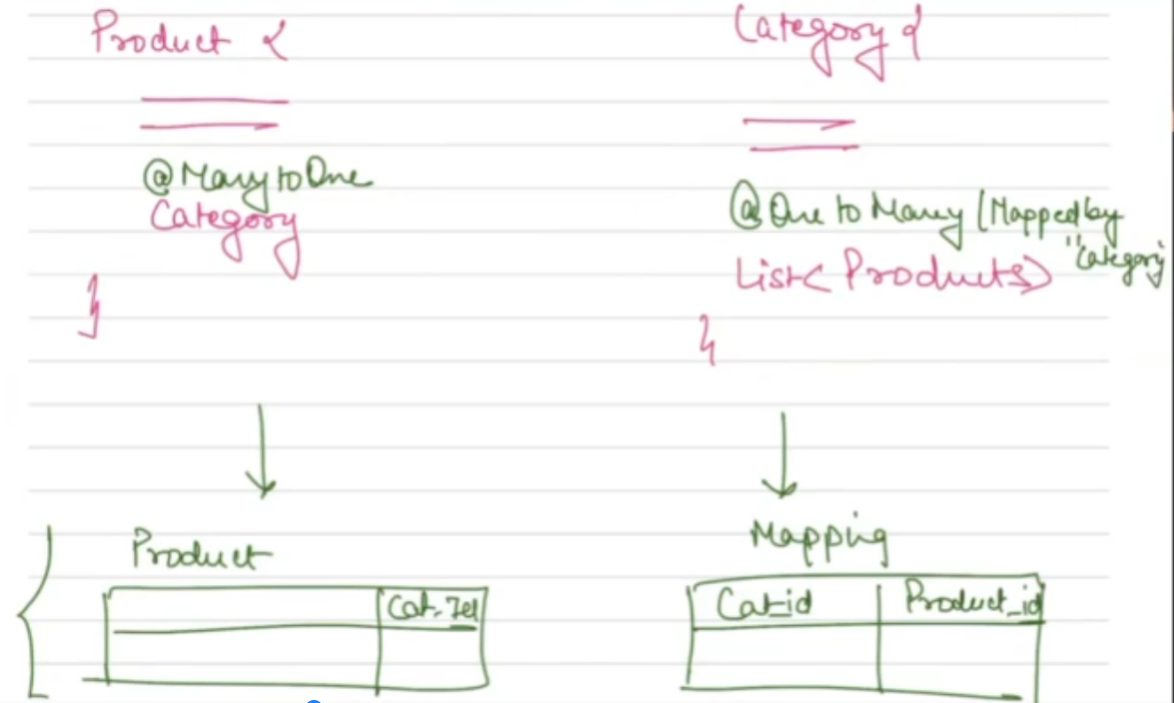
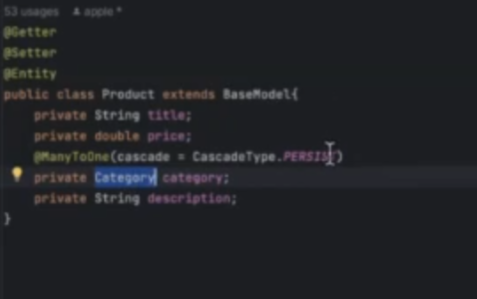


Remember exception handling works in a way that the more specific way an exception is handled accordingly that piece of code will be called.

*   
  If we have an outer object and inside it another object which is not a collection then there ia eager initialization by default where details of both the objs would be fetched at the same time.

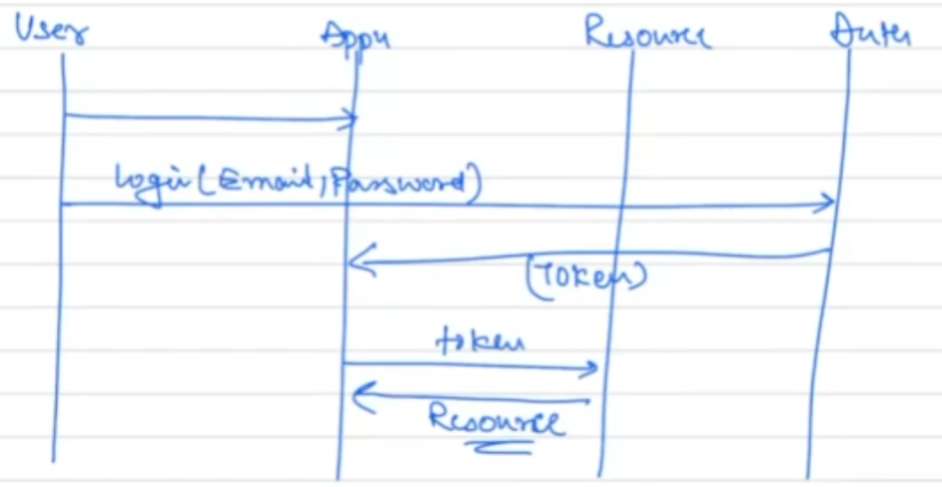
Whereas if there a collection of obj inside an outer obj then by default there is lazy initialization.

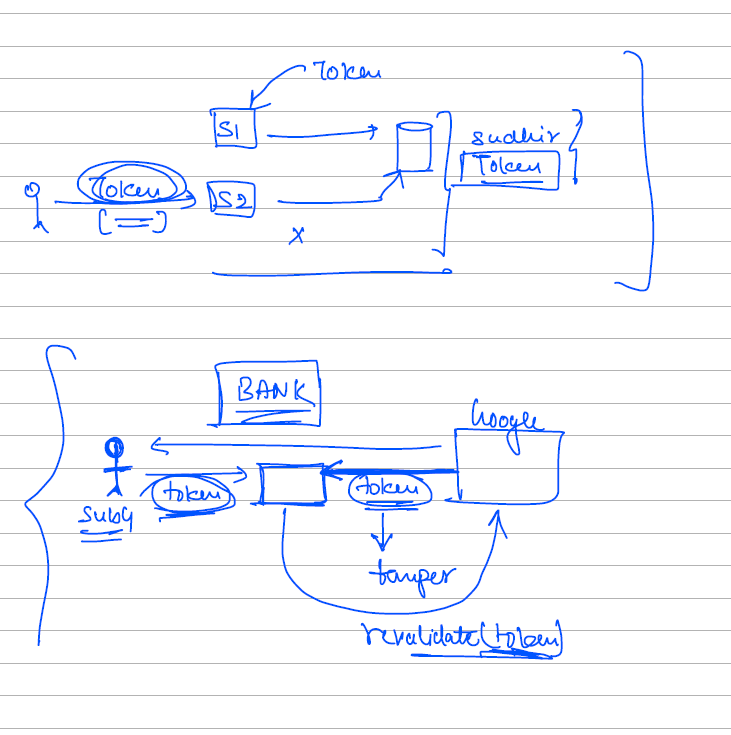
The cardinality betw produc and category is m:1. In this relation the foreign key will be present in the many side ie: FK will be present in the many side.

*   
    
  Here we see that a duplicate reln being createad. Considering each category can have many products when we create a table we see a duplicate reln being created. In order to handle such scenario we use something as MappedBy Category so that we do not create 2 tables with duplicate reln.
*   
    
  We have diff cascading types. Now if a category is not present then the PERSIST type cascading will simply create the category we are passing while creating a new product. Adv being we do not have to write a specific logic to save the category separately.
* It is very imp to maintain DB change history. One is Flyway and the other is LiquiBase.

**Authentication and Authorization**

* We use tokens so that for every request we do not have to send a username and pwd. Besides this even for a generated token we need to validate it if the token belongs to the right user making request, and in order to avoid a DB call in this situation we use something as self validated token.
* Now OAuth is like a framework which in itself has all the details of. Now consider the below diagrams:





Now suppose user logs into server S1 where he uses google authentication now he would be redirected to gmail login page. Now here you enter your credentials using which a token is generated. Here the google Authorization server will generate the token and this token is then transferred to me suppose say xyz server (Resource server) and now this token would also be saved at a central data base as we will have multiple servers to which the Load Balancer can forward a request to. Also this stored token can be used for any auditing purpose.  
  
Only systems which take security as top most priority use the tokens saved in the db as a part of the validation for each and every request. Whereas in the normal scenario this token is stored in the cookies of my browser and has all kinds of details as a part of it like:

1. The ip address logged in from.
2. The username
3. Expiry time
4. Deleted or not

Now using the token alone a request can be validated as it has all the details and we can choose not to validate it from the db.  
  
Note:

  
Encoding can be any strategy like Base64. But since this alone cannot be used we add a secret key.



**Public IP vs. Static IP: Easy Explanation**

**1. Public IP:**

* **What is it?** A **public IP address** is an IP address that is assigned to a device (like a computer, router, or server) to be accessible over the internet. It's unique across the whole internet.
* **When is it needed?** You need a **public IP** when you want a device or service to be reachable from outside your local network, like a website server or online gaming console.
* **Example**: Think of a public IP like a **home address**. Anyone in the world can send you mail if they know your address. Similarly, if your device has a public IP, anyone on the internet can interact with it (like visiting a website you host).

**Example Use**: Hosting a website on a server so users can visit it from anywhere in the world.

**2. Static IP:**

* **What is it?** A **static IP address** is an IP address that doesn’t change. It’s permanently assigned to a device, ensuring that it always has the same address, even after rebooting or disconnecting.
* **When is it needed?** You need a **static IP** when you require a constant IP address, such as for services like remote access, web hosting, or email servers. It ensures that users or devices can always connect to the same IP address without changes.
* **Example**: Think of a static IP like a **fixed phone number**. No matter when someone tries to call you, the number remains the same. Similarly, with a static IP, the address of the device stays constant.

**Example Use**: A company hosting a VPN service so employees can always connect to the same IP address to access company resources.

**Key Difference:**

* A **public IP** allows you to connect to the internet, while a **static IP** is a type of public IP (or local IP) that remains fixed. Not all public IPs are static—many change periodically (called dynamic IPs).

**When You Need Each:**

* **Public IP**: If you want to make your device accessible to the broader internet, like a website or game server.
* **Static IP**: If you need a constant, never-changing IP address for things like hosting servers, remote access, or VPN services

**AWS**

When you connect to a **public IP** of an EC2 instance, your request is redirected to the instance's **local (private) IP** through a process known as **Network Address Translation (NAT)**. Here's how this works in the context of AWS:

**1. Public Subnet and Elastic Network Interface (ENI)**

* When an EC2 instance is launched in a **public subnet** (within a VPC), AWS assigns it both a **public IP** (for external communication) and a **private IP** (for internal communication within the VPC).
* The EC2 instance is associated with an **Elastic Network Interface (ENI)**, which is the network adapter that holds both the public and private IPs.

**2. Internet Gateway**

* To allow instances in a public subnet to communicate with the internet, AWS uses an **Internet Gateway (IGW)**. This is a VPC component that enables internet traffic to flow to and from instances.
* The Internet Gateway performs **NAT** between the public and private IP addresses of instances.

**3. Network Address Translation (NAT) Process**

* When you try to connect to an EC2 instance via its **public IP** (e.g., using SSH), the following steps occur:
  1. **Client Request**: From your local machine, you initiate a request to the instance's **public IP address**.
  2. **Internet Gateway (NAT)**: The request reaches the **Internet Gateway**, which knows that the public IP is mapped to a specific private IP of the instance within the VPC.
  3. **Redirection to Private IP**: The Internet Gateway translates the **public IP** into the **private IP** and forwards the request to the EC2 instance over the VPC’s internal network.
  4. **Response**: The EC2 instance responds using its private IP, and the Internet Gateway translates the private IP back into the public IP when sending the response back to the client.

This NAT process is seamless and transparent to the user, allowing external clients to access the instance using the public IP while the instance operates within the private VPC using its private IP.

**Example Flow:**

1. **You connect via SSH**:
   * Command: ssh -i "your-key.pem" ec2-user@54.67.102.10
   * 54.67.102.10 is the public IP of the instance.
2. **Request reaches Internet Gateway**:
   * The request travels through the internet and reaches the VPC’s Internet Gateway.
3. **NAT Process**:
   * The Internet Gateway translates 54.67.102.10 (public IP) to the instance's private IP, e.g., 10.0.0.5.
4. **EC2 Instance Response**:
   * The instance processes the request and responds, sending data back to the Internet Gateway.
   * The Internet Gateway translates the private IP (10.0.0.5) back to the public IP (54.67.102.10) before forwarding the response to your client.

**Key Concepts:**

* **Internet Gateway (IGW)**: A managed service that allows instances in public subnets to communicate with the internet.
* **NAT (Network Address Translation)**: The process of mapping a public IP address to a private IP address, which happens at the Internet Gateway level in AWS.
* **Public Subnet**: A subnet that has a route to the Internet Gateway, allowing public IP addresses to function.
* **Private Subnet**: A subnet without internet access, where instances only have private IPs and cannot be accessed directly from the internet.

**Important Notes:**

* **Elastic IP**: If you want a **static public IP** that persists even if the instance is stopped and started, you can associate an Elastic IP with the instance.
* **Security Groups and Network ACLs**: Even though your instance is publicly accessible via a public IP, it is still protected by **Security Groups** and **Network Access Control Lists (ACLs)**, which act as firewalls, ensuring that only authorized traffic can reach the instance.

**Diagram:**

scss

Copy code

[ Your Computer ]

|

Public IP (54.67.102.10)

|

Internet Gateway (NAT) [VPC]

|

Redirects Public IP to Private IP

|

Private IP (10.0.0.5) -> EC2 Instance

In summary, when you connect via the public IP, AWS’s Internet Gateway handles the translation of the public IP to the instance's private IP using NAT, allowing seamless communication between external clients and internal resources.

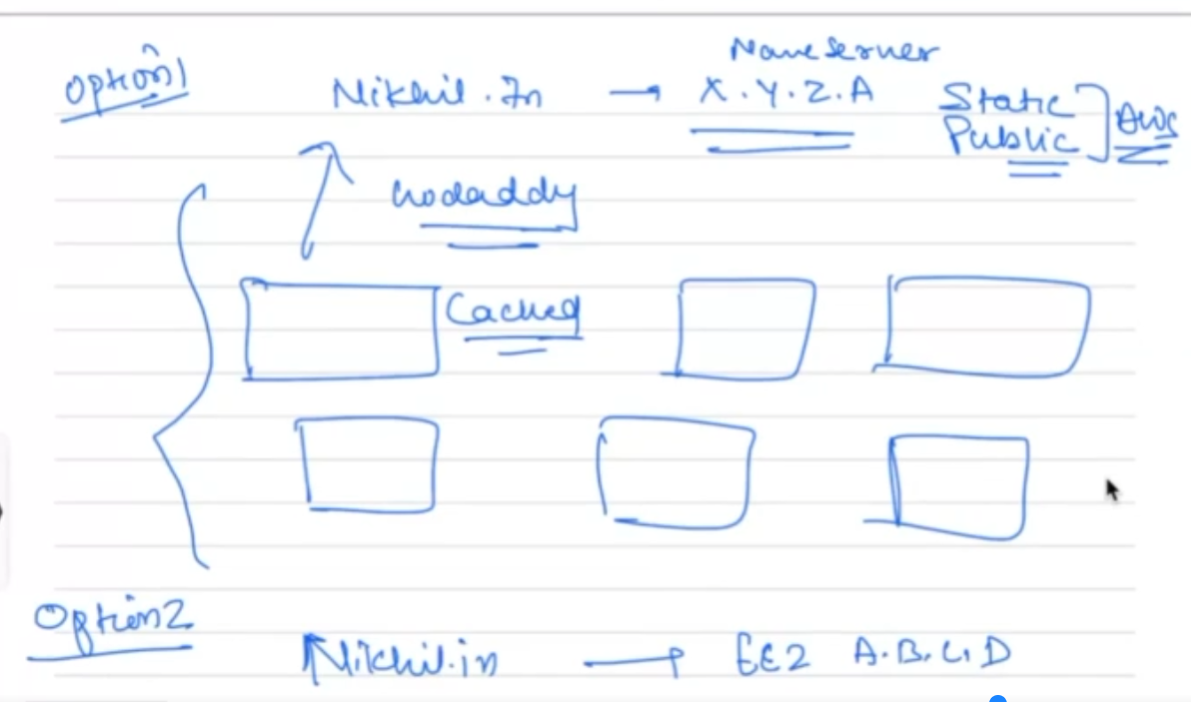
If you want local db data to be put in RDS database, first export local db dump and upload to S3. Then from S3 restore this data to rds database.

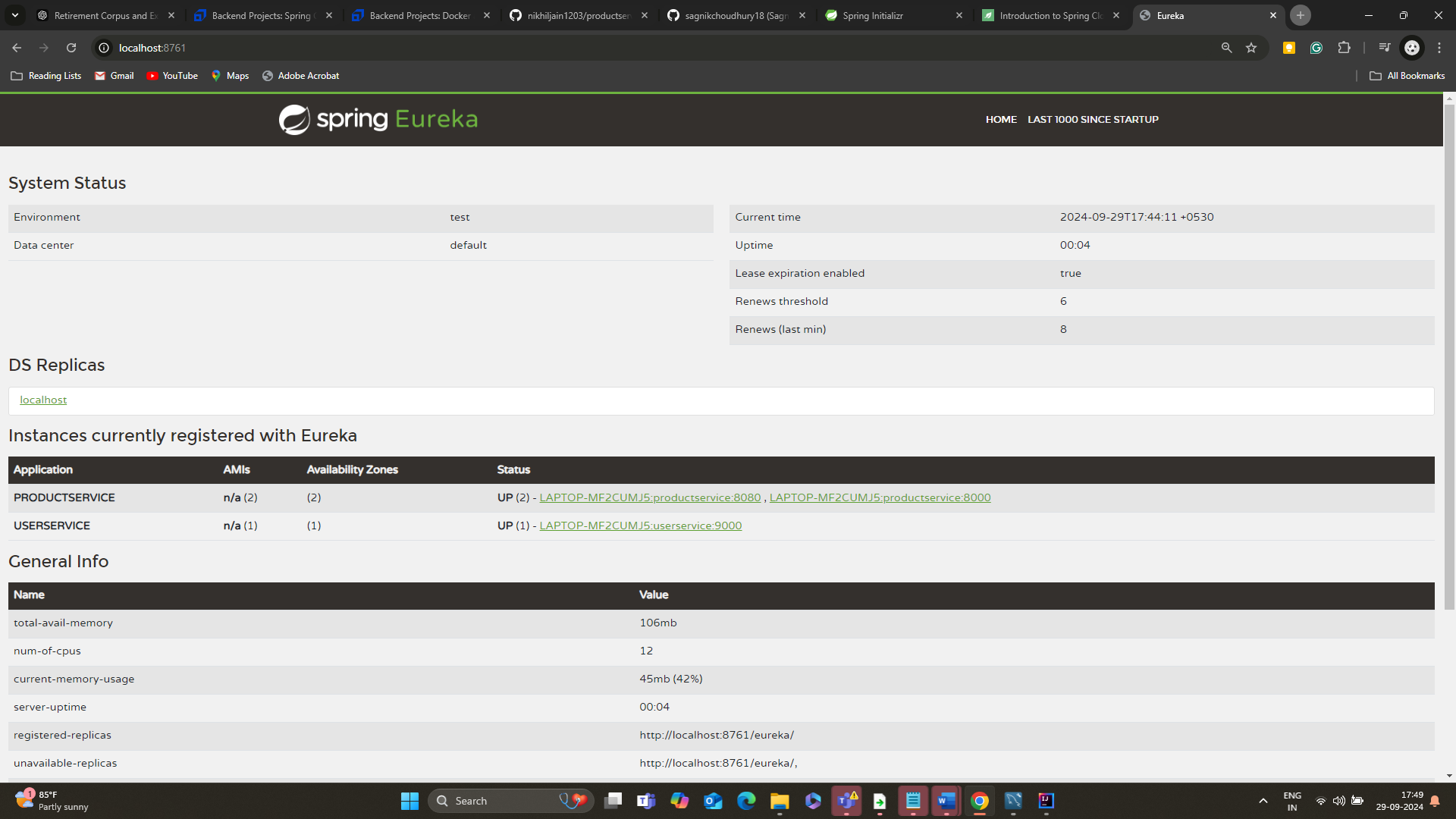
**R53**

Option 1

Suppose I have an app which is famous and has its public static ip address cached in browser or somewhere. Then using this cached value my DNS reoln is done.

Option 2  
  
But suppose my app is hosted in EC2 instance now this ip address can change. Now to know the ip address AWS provide us with Route53 where we have a Named server which stores the mapping betw a domain name and an ip address. This would act like a map and help in DNS resolution.





The load balancing is done using the RestTemplate where we use the @LoadBalancing annotation.  
  
If we have separate services which needs to be loadbalanced separately we should have separate rest templates for each service. Normally when we add @LoadBalanced annotation at the RestTemplateLevel, whichever call we make be it for a User or FakeStore to get the products it would be expected to register them as a sperate service. Therefore it is better to configure separate RestTemplate for each service.

If suppose we have a scenario where UserService needs to be called from within ProductService we use something like :  
UserDto userDto = restTemplate.getForObject("http://userservice/users/1", UserDto.class);  
Here irrespective of the number of userService instance that runs since we have the @LoadBalance at the restTemplate level based on the registered application name the request would automatically be forwarded to one of the instances with userService running.  
  
  
  
**APIGATEWAY**

Any call coming through the APIGATEWAY would go to all the services. For eg if ProductService has 3 instances running on 3 different ports. Now any request coming in would go to all the 3 instances. The ApiGateWay based on the predicate identifies which service needs to be called.

