



9 Feb 2024


Backend Projects:
Finishing API's






Session Recording

Backend Projects: Finishing API's

 9:00 PM Fri, 9 Feb

Watch Recording




Day 196 - Backend Projects: Finishing API's


Fri, 9 Feb 2024

Mandatory

Session 12.52%

Help Requests

Feedback 

 Your Expert Mock Interview is pending

Agenda : High Level Design

[Learn more about Mock Interviews](#)

Nikhil Jain10:53 PM

<https://protobuf.dev/>

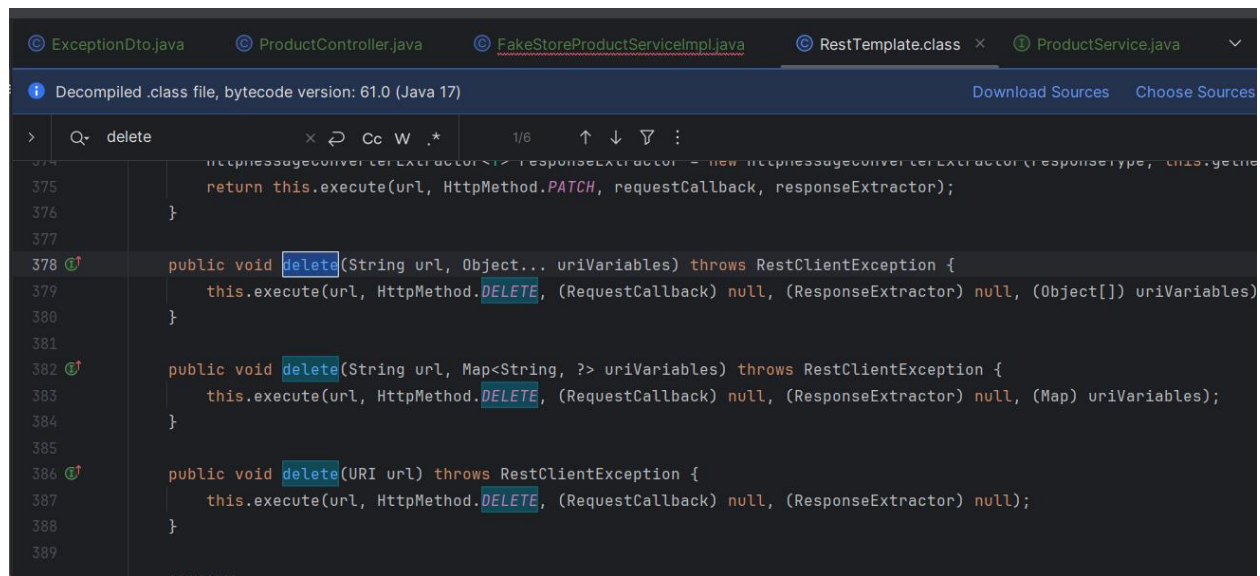
Nikhil Jain9:46 PM

<https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RequestCallback.html>

Delete a product API:

```
@Override
public void deleteProductById() {
}
```

We need by id.. pass id in method. We will use restTemplate. Delete via restTemplate..

A screenshot of an IDE window showing the decompiled source code for the RestTemplate class. The window title bar includes tabs for ExceptionDto.java, ProductController.java, FakeStoreProductServiceImpl.java, RestTemplate.class (selected), and ProductService.java. Below the tabs, a status bar indicates 'Decompiled .class file, bytecode version: 61.0 (Java 17)'. The main editor area displays three overloaded delete methods. The first method (line 378) is public void delete(String url, Object... uriVariables) throws RestClientException. The second method (line 382) is public void delete(String url, Map<String, ?> uriVariables) throws RestClientException. The third method (line 386) is public void delete(URL url) throws RestClientException. All methods use this.execute(url, HttpMethod.DELETE, ...) to perform the delete operation. The word 'delete' in the method signatures is highlighted in blue, and 'DELETE' in the execute calls is highlighted in red. The line numbers 374 through 389 are visible on the left margin.

Delete needs url, object..... url variables..

A screenshot of an IDE window showing the implementation of the deleteProductById method. The window title bar includes tabs for ExceptionDto.java, ProductController.java, FakeStoreProductServiceImpl.java, RestTemplate.class, and ProductService.java. Below the tabs, a status bar indicates 'no usages new *'. The main editor area displays the implementation of the deleteProductById method. The method signature is public void deleteProductById(Long id) {. The implementation uses RestTemplate restTemplate = restTemplateBuilder.build(); and ResponseEntity<> responseEntity = restTemplate.delete(genericProductUrl, ...);. The line numbers 49 through 54 are visible on the left margin.

But delete is a void function, returns nothing.. so we don't need to add anything here. In left side. Also we add 2 arg in method.... url and id.

```

no usages new *
@Override
public void deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    restTemplate.delete(genericProductUrl, id );
}

```

Override has problem.. means the interface definition and other implementation needs to be corrected...

```

7
5 usages 2 implementations new * 1 related problem
8 public interface ProductService {
    1 usage 2 implementations new *
9     Product getProductById(Long id) throws ProductNotFoundException;
    1 usage 2 implementations new *
10    List<Product> getAllProducts();
11
    no usages 1 implementation new * 1 related problem
12    void deleteProductById(Long id);
    1 usage 2 implementations new *
13    Product addProduct(Product product);
    no usages 2 implementations new *
14    void updateProductById();
15 }

```

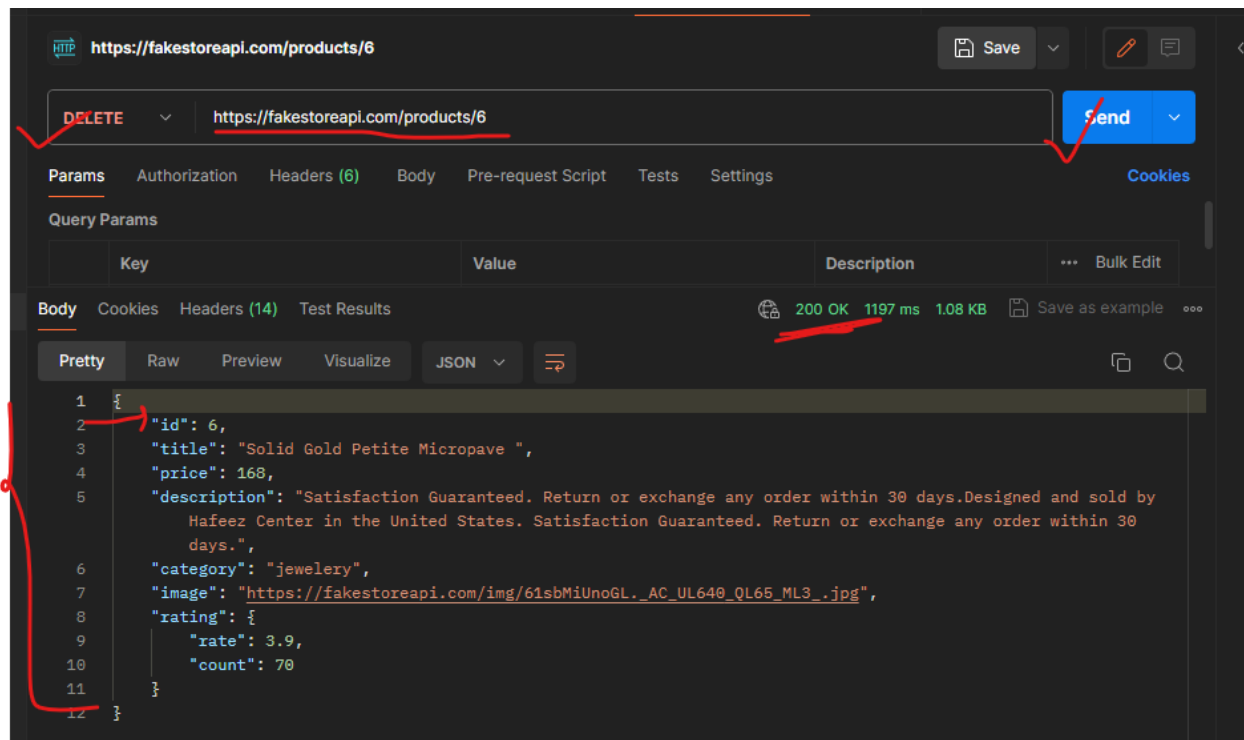
```

new *
7  @Service("SelfProductService")
8  public class ProductServiceImpl implements ProductService{
    1 usage new *
9      @Override
10     public Product getProductById(Long id) {
11
12         return null; // null for now
13     }
    1 usage new *
14     @Override
15     public List<Product> getAllProducts() { return null; }
18
    no usages new *
19     @Override
20     public void deleteProductById(Long id) {
21     }
    1 usage new *

```

Lest run and see ,,

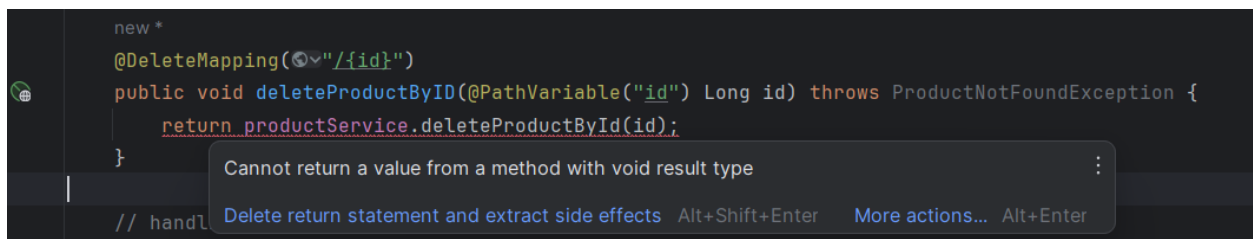
First lets see how fakestore product API doing it:



For confirmation it is returning the object which it is deleting..

This is How fakestore is confirming..

We also have to write the delete mapping in controller:



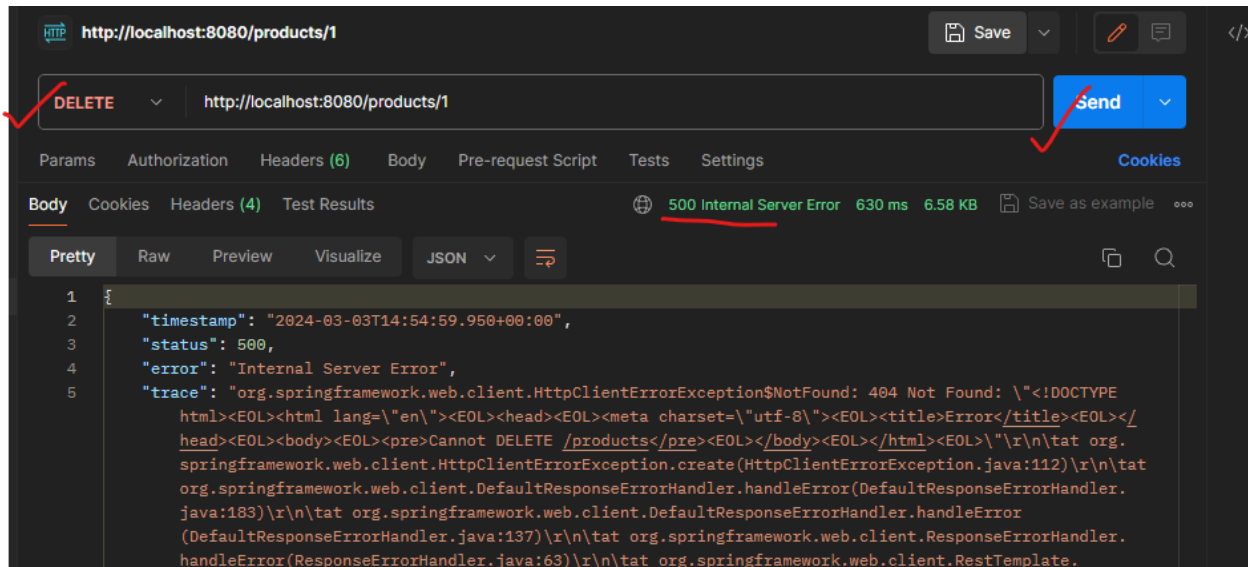
```

@DeleteMapping("/{id}")
public void deleteProductByID(@PathVariable("id") Long id) throws
ProductNotFoundException {
    productService.deleteProductById(id);
}

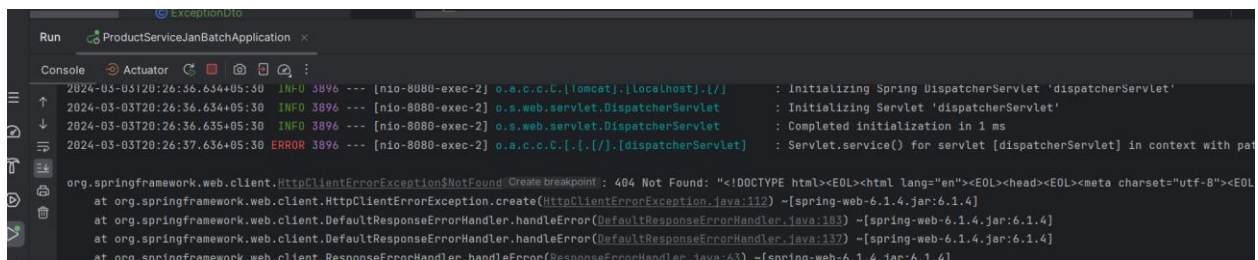
```

We will not have a return type as delete has void.. also as it's a void. Return staemenet will not be there.... Just method clal of productservice IF.

Now lets try to delete form our side..



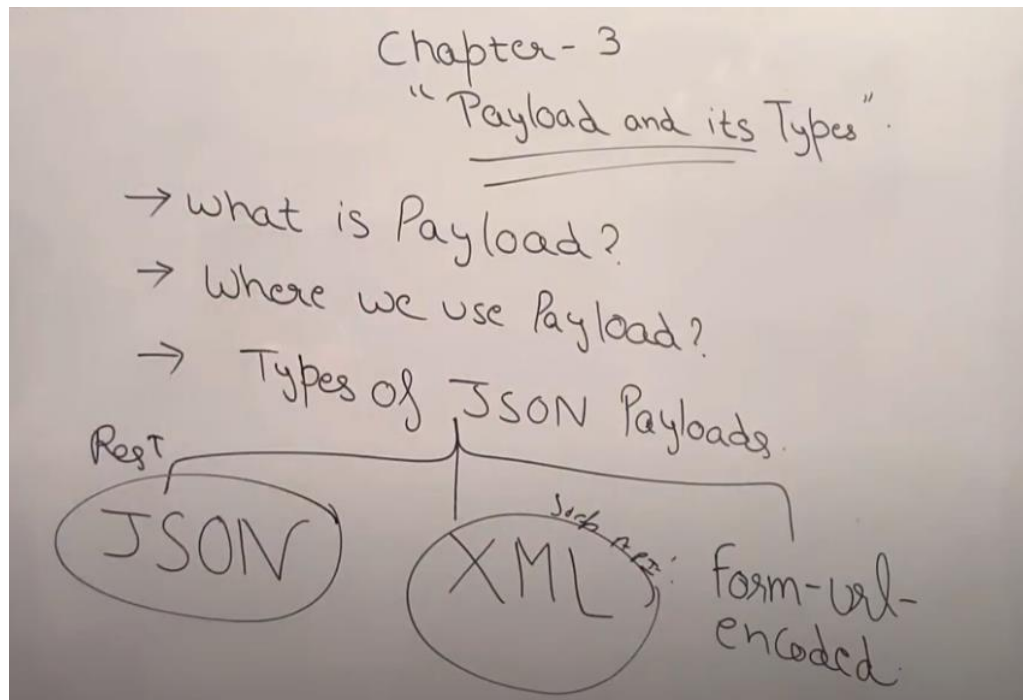
"org.springframework.web.client.HttpClientErrorException\$NotFound: 404 Not Found:



are we sending any payload in postman

PAYLOAD: http request me body hota hai. Body ko payload bolte hai.

https://www.youtube.com/watch?v=6Z6-uF_l2n0&ab_channel=LearnTestingByAD



Json used for rest, XML used for SOAP.

API testing me.. body ko hi payload bolte hai. Body of API = payload.

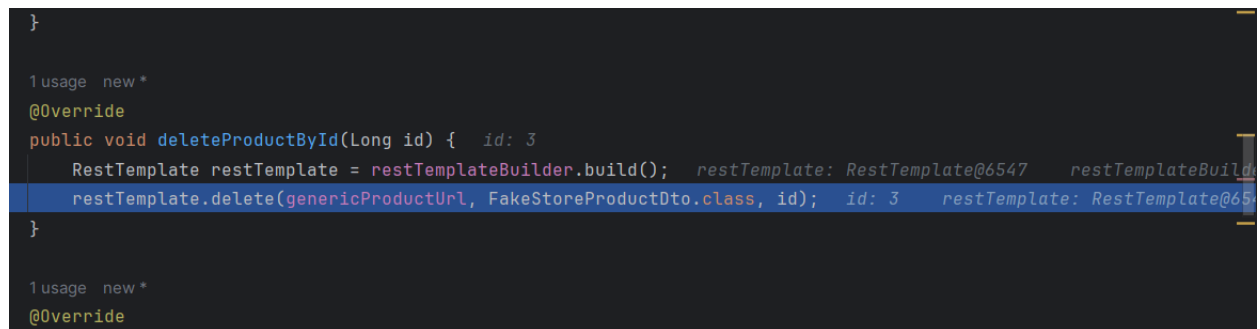
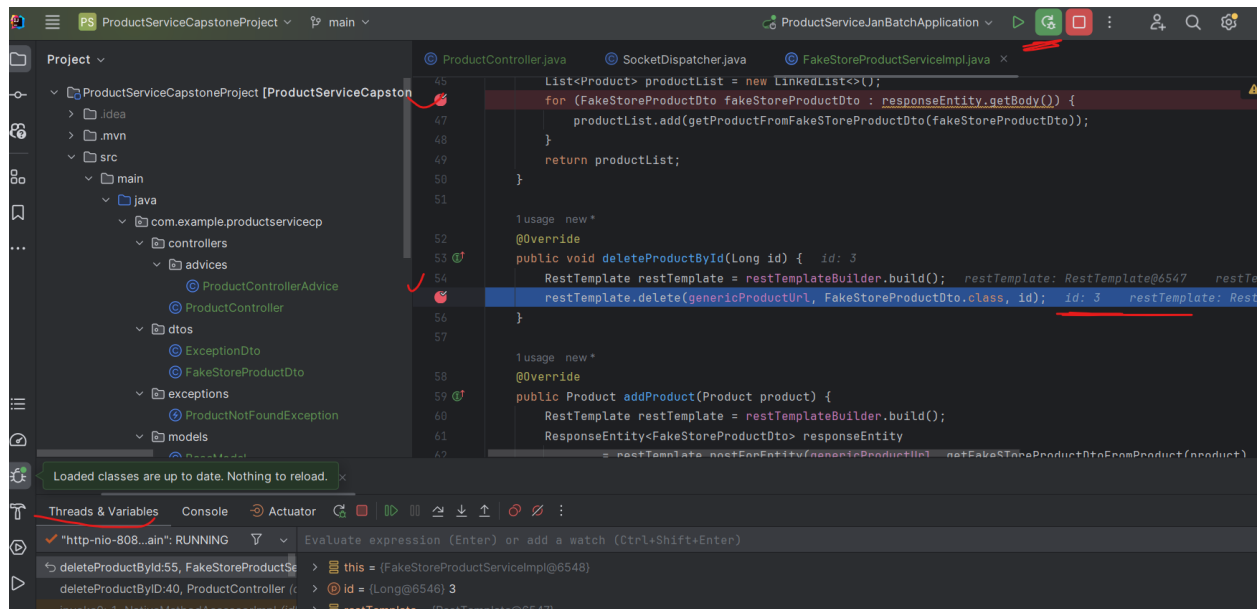
What is payload?

1. A payload is the body of HTTP request or response.
2. When browsing the web browser request or receive an HTML payload..
3. Typically when working with HTTP api we will send/receive JSON or XML payload

Why Payload is used?

1. To send or received data to server...
2. Simply used to differentiate information which is used to perform any action and information only for support..

To send or received data uska format ko payload bolte hai..



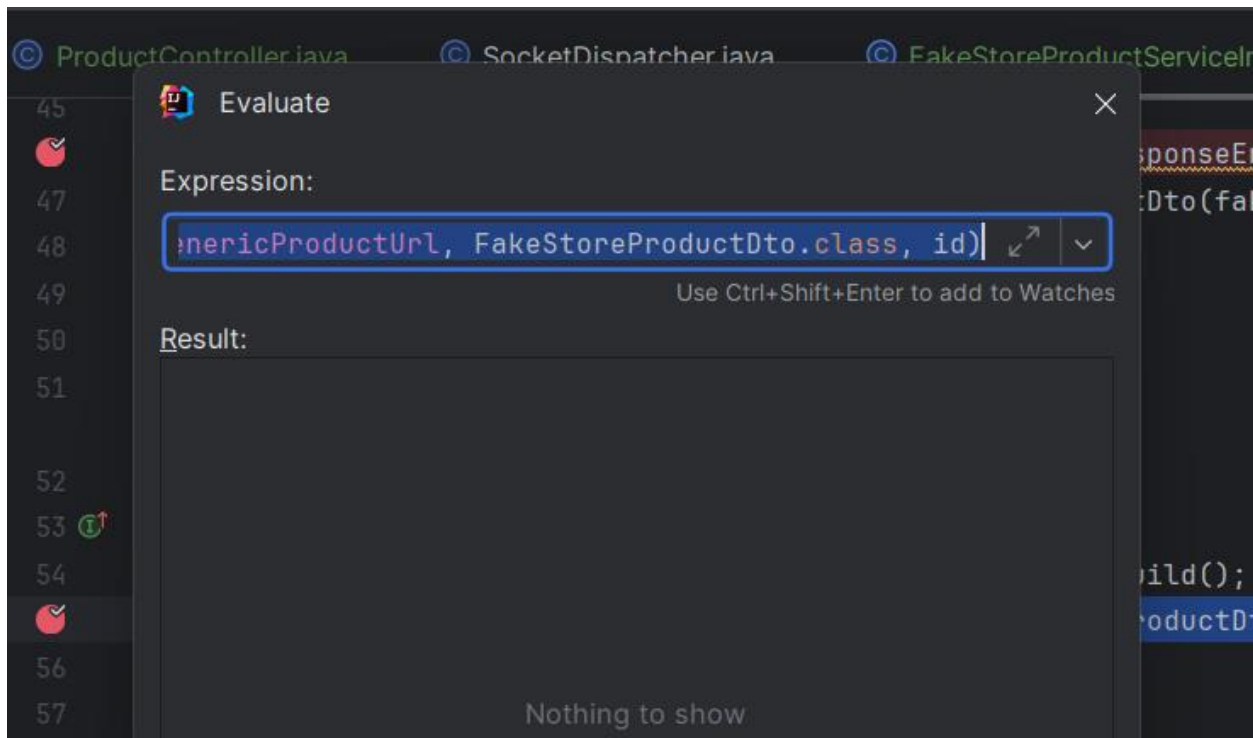
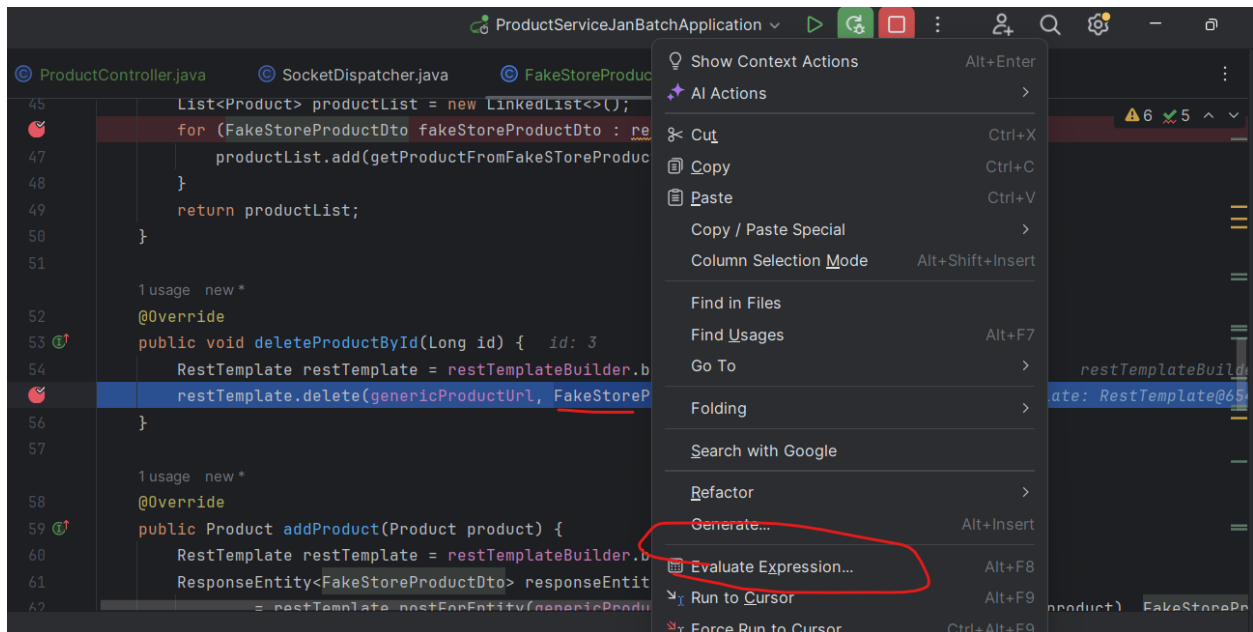
Id = 3... aya idhar..

We are getting the id.. then there must be issue in **FakeStoreProductDto.class**, so we will evaluate the expression///

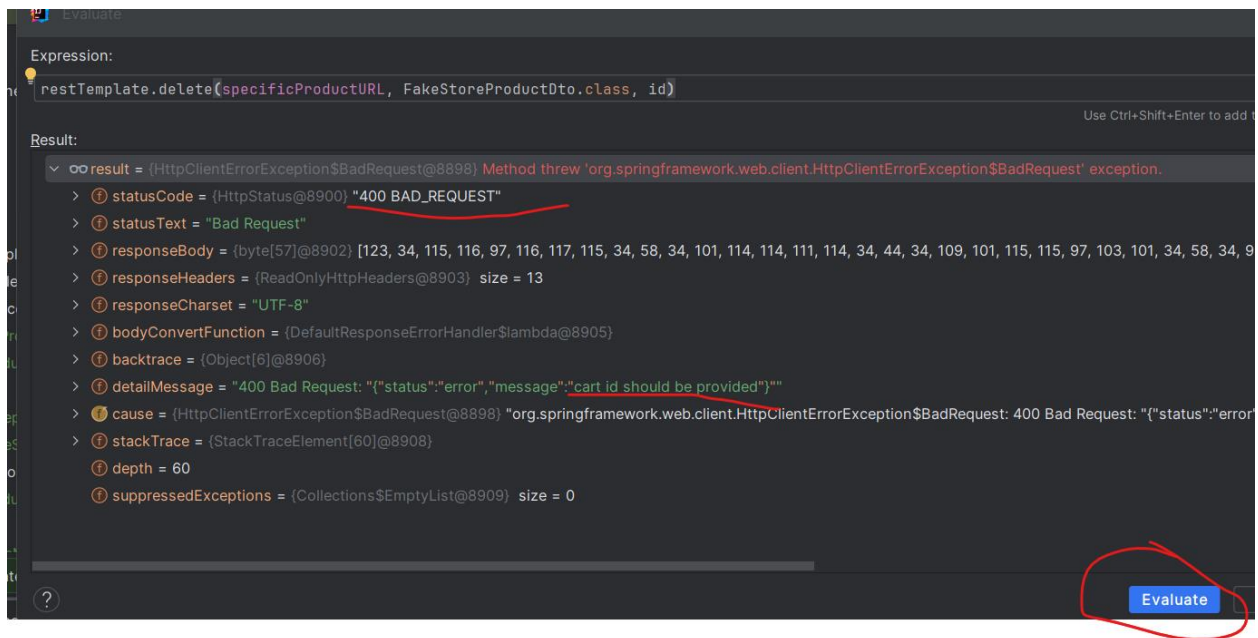
To do so select the entire line/ entire expression:

```
restTemplate.delete(genericProductUrl, FakeStoreProductDto.class, id);
```

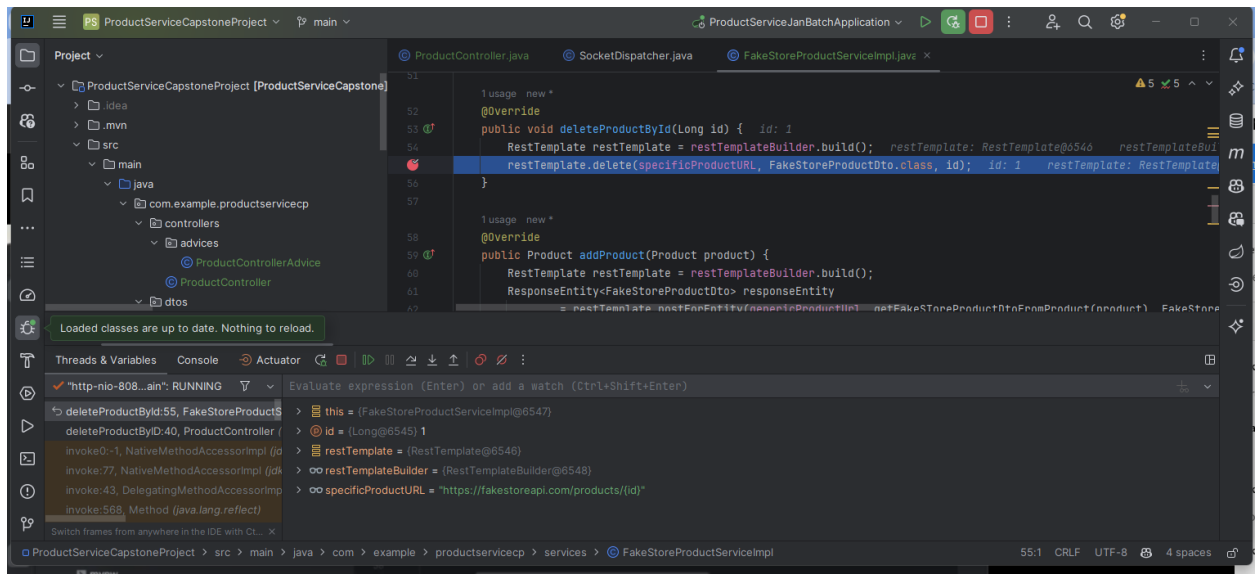
right click evaluate expression..



Click on evaluate:



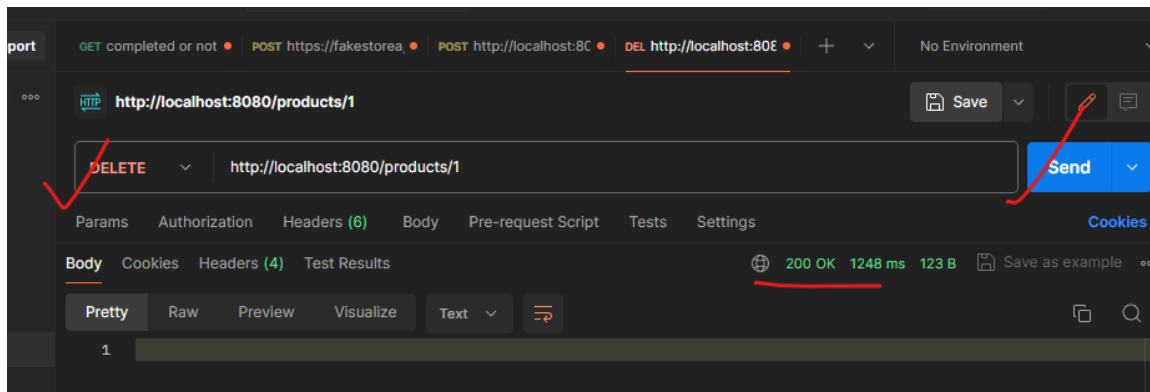
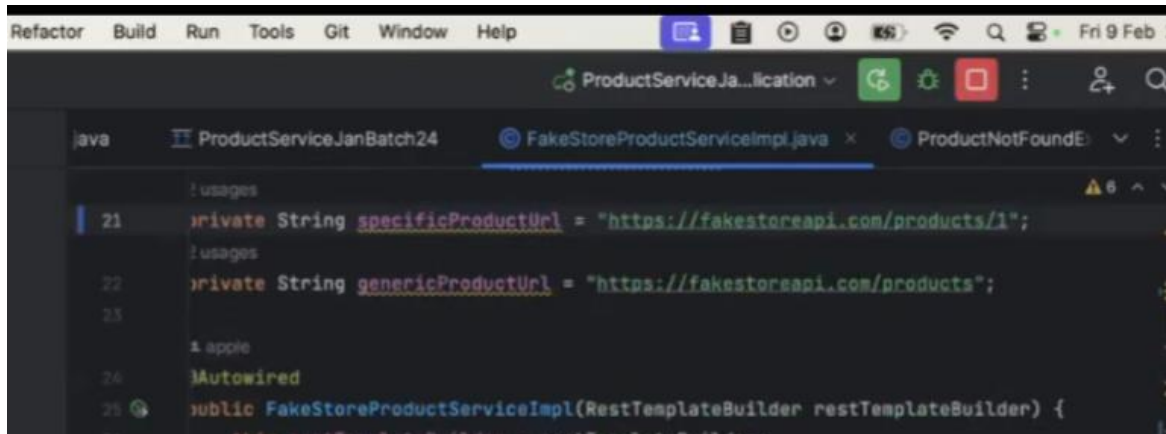
Its asking for a cart id..



Problem is whatever info we are sending is not enough... its need cart id... asking for cart id..

Let's call it hardcoded way:

```
private String specificProductURL = "https://fakestoreapi.com/products/1";
```



We are now getting 200 okay.

when we are sending id, its not able to append the id directly.. so some problem with delete function.



In controller also we update:

```
@DeleteMapping("/{id}")
public void deleteProductById(@PathVariable("id") Long id) throws
ProductNotFoundException {
    productService.deleteProductById(id);
}
```

```

576     }
577
578     public void delete(String url, Object... uriVariables) throws RestClientException {
579         this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null, (Object[])
580     }
581
582     public void delete(String url, Map<String, ?> uriVariables) throws RestClientException {
583         this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null, (Map) uriV
584     }
585
586     public void delete(URI url) throws RestClientException {
587         this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null);
588     }

```

Removed the hardcoded id:

```
private String specificProductURL = "https://fakestoreapi.com/products/{id}";
```

some issue with delete function:

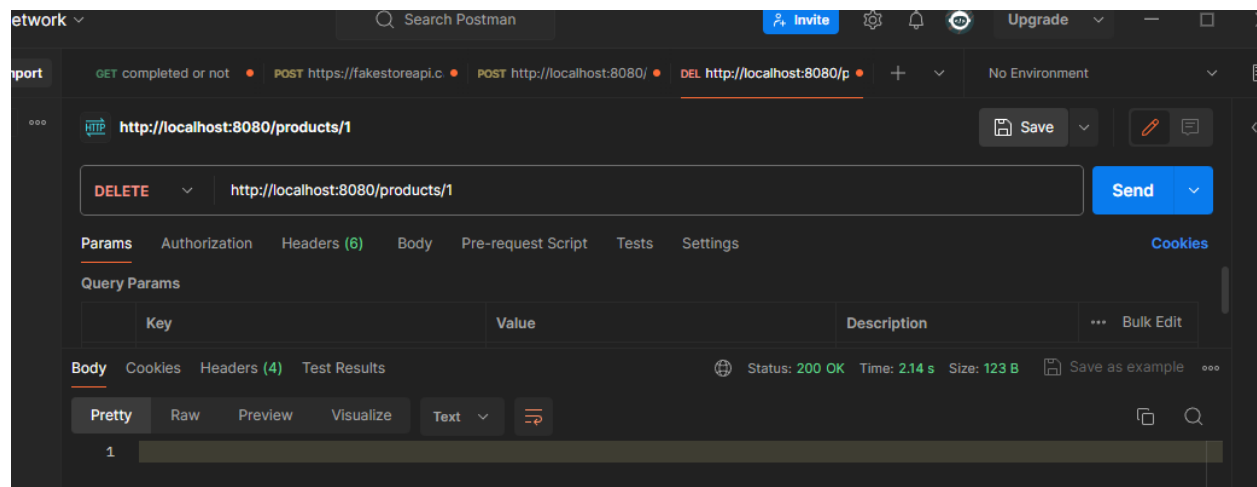
We have deleted the `fakestoredto.class`. the DTO is not needed. Added just id here...

```

52     1 usage new *
53     @Override
54     public void deleteProductById(Long id) {
55         RestTemplate restTemplate = restTemplateBuilder.build();
56         restTemplate.delete(specificProductURL, id);
57     }

```

Now fine..



The fakestore api gives which object is deleted. But my API has no post validation.. I can't verify. But if we use directly fakestore... we can do this. Resttemplate doesn't give this facility.

The delete function has 3 impl:

```
public void delete(String url, Object... uriVariables) throws
RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback)null,
(ResponseExtractor)null, (Object[])uriVariables);
}

public void delete(String url, Map<String, ?> uriVariables) throws
RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback)null,
(ResponseExtractor)null, (Map)uriVariables);
}

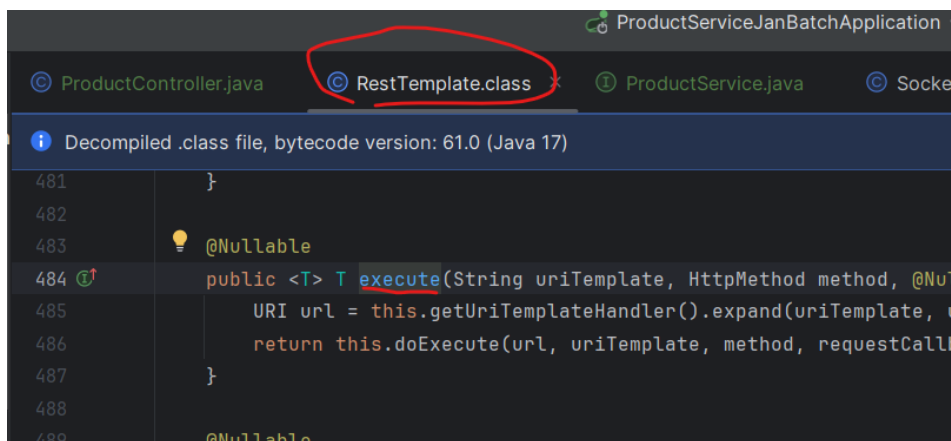
public void delete(URL url) throws RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback)null,
(ResponseExtractor)null);
}
```

```
public void delete(String url, Object... uriVariables) throws RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null, (Ob
}

public void delete(String url, Map<String, ?> uriVariables) throws RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null, (Map
}

public void delete(URL url) throws RestClientException {
    this.execute(url, HttpMethod.DELETE, (RequestCallback) null, (ResponseExtractor) null);
}
```

The delete methodWe have this.execute.. its using execute function of restTemplate class:



The screenshot shows an IDE window with the title 'ProductServiceJanBatchApplication'. The tab bar at the top contains 'ProductController.java', 'RestTemplate.class' (which is selected and circled in red), 'ProductService.java', and 'Socket'. Below the tab bar, a message bar indicates 'Decompiled .class file, bytecode version: 61.0 (Java 17)'. The code editor displays the 'execute' method of the 'RestTemplate' class, starting at line 481. The method signature is 'public <T> T execute(String uriTemplate, HttpMethod method, @Nullable'. The code includes a call to 'this.getUriTemplateHandler().expand(uriTemplate, ...)' and a return statement 'return this.doExecute(url, uriTemplate, method, requestCalll'. The method ends at line 487 with a closing brace. Line 489 shows the start of another method with the '@Nullable' annotation.

I cant verify. But if we use directly fakestore... we can do this. Resttemplate doesn't give this facility.

```
@Override
public ResponseEntity<FakeStoreProductDto> deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
    =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    return restTemplate.execute(specificProductURL, HttpMethod.DELETE,
requestCallback, responseExtractor, id);
}
```

```
1 usage new *
@Override
public ResponseEntity<FakeStoreProductDto> deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback = restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
        = restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    return restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback, responseExtractor, id);
}
```

Client need product not dto

```
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
    =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

```

1 usage new *
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback = restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
        = restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    ResponseEntity<FakeStoreProductDto> responseEntity = restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback, responseExtractor);
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}

```

Now since we are returning product, update everywhere:

```

36
37
38     new *
39     @DeleteMapping("/{id}")
40     public Product deleteProductById(@PathVariable("id") Long id) throws ProductNotFoundException {
41         return productService.deleteProductById(id);
42     }

```

```

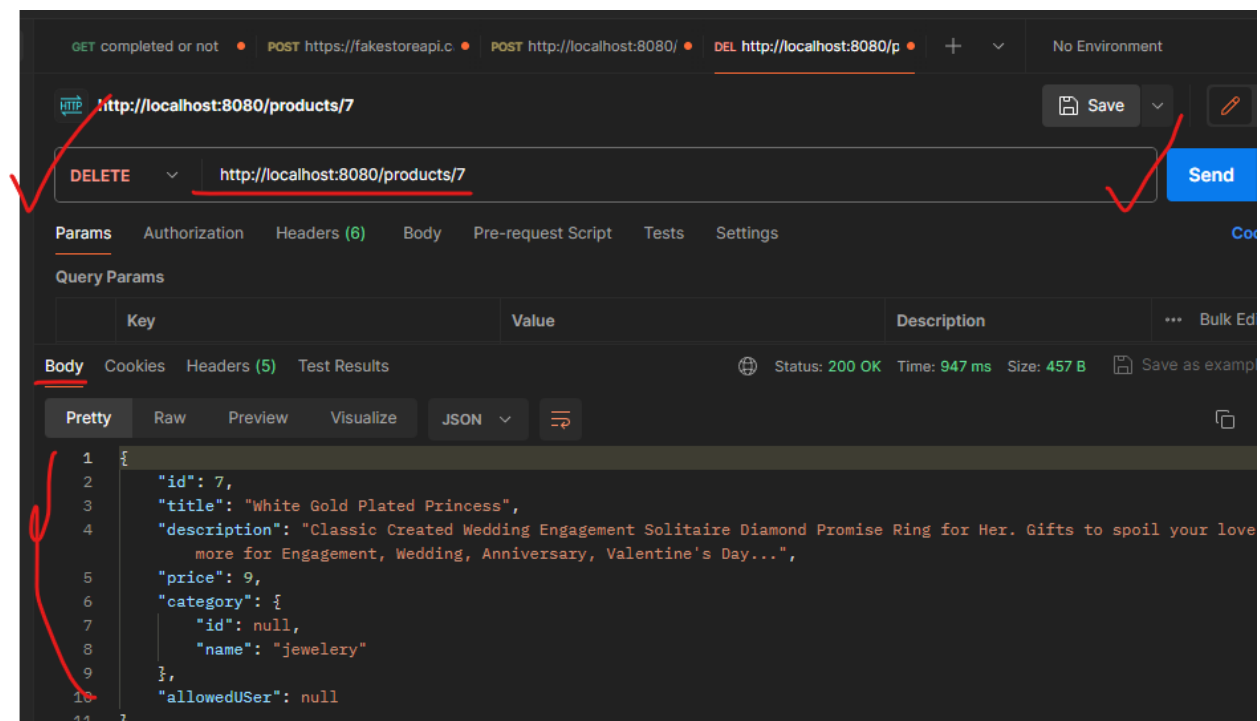
1 package com.example.productservicecp.services;
2
3 > import ...
4
5 5 usages 2 implementations new *
6
7
8 public interface ProductService {
9     1 usage 2 implementations new *
10     Product getProductById(Long id) throws ProductNotFoundException;
11     1 usage 2 implementations new *
12     List<Product> getAllProducts();
13
14     1 usage 2 implementations new *
15     Product deleteProductById(Long id);
16
17     1 usage 2 implementations new *

```



```
ProductServicesanBatchApplication
ProductController.java ProductServiceImpl.java ProductService.java
11
12     return null; // null for now
13 }
14
15 @Override
16 public List<Product> getAllProducts() { return null; }
17
18
19 @Override
20 public Product deleteProductById(Long id) {
21     return null;
22 }
```

Now let's run and test:



can we make a helper method for these?

Will come

What we had done to get deleted verification:

We have seen that restTemplate.delete don't give us body:

```
// restTemplate.delete(specificProductURL, id);
```

It doesn't give us a function which can return us a response entity.

We can go and locate the class of restTemplate. And try to identify what's happening.. we can locate a function "getForEntity" inside restTemplate class."

```
public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType,
Object... uriVariables) throws RestClientException {
    RequestCallback requestCallback =
this.acceptHeaderRequestCallback(responseType);
    ResponseExtractor<ResponseEntity<T>> responseExtractor =
this.responseEntityExtractor(responseType);
    return (ResponseEntity) nonNull((ResponseEntity) this.execute(url,
HttpMethod.GET, requestCallback, responseExtractor, uriVariables));
}
```

```
2 usages
public <T> ResponseEntity<T> getForEntity(String url, Class<T> responseType, Object... uriVariables) throws Rest
    RequestCallback requestCallback = this.acceptHeaderRequestCallback(responseType);
    ResponseExtractor<ResponseEntity<T>> responseExtractor = this.responseEntityExtractor(responseType);
    return (ResponseEntity) nonNull((ResponseEntity) this.execute(url, HttpMethod.GET, requestCallback, response
```

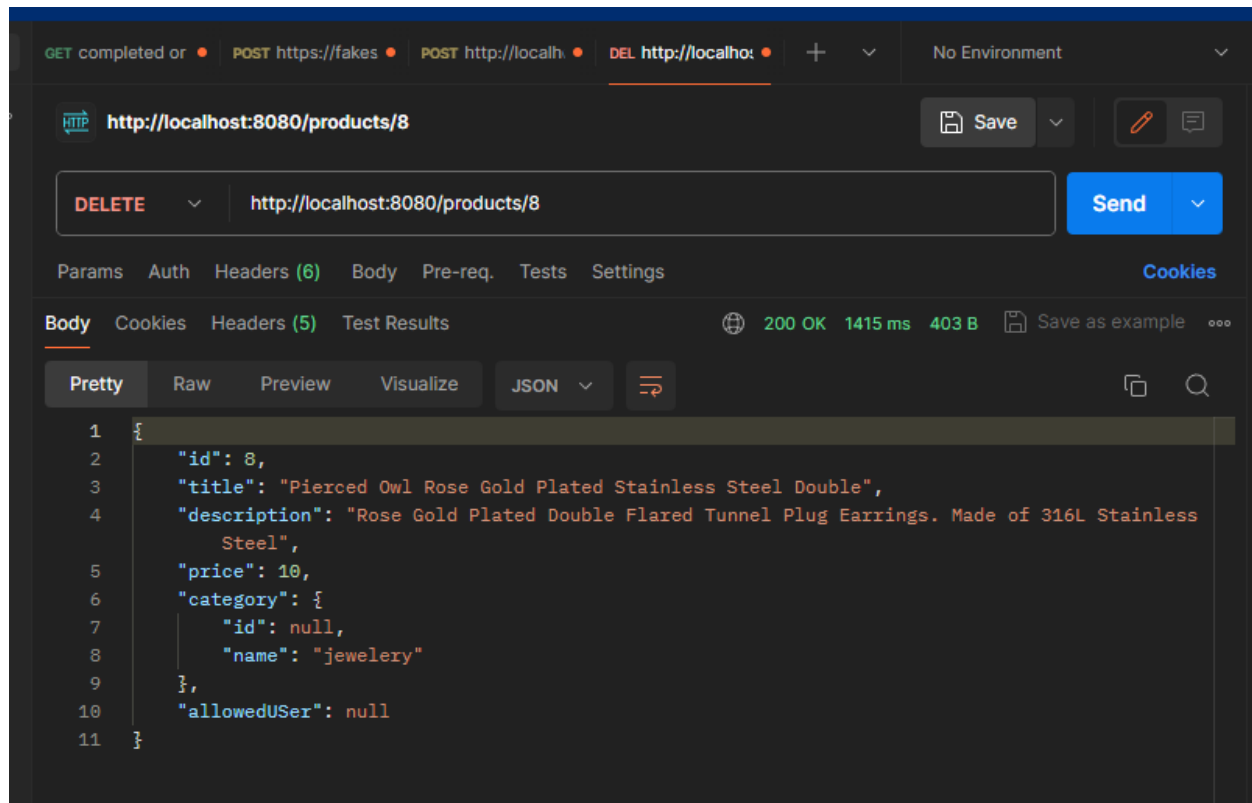
This getForEntity is very similar with delete. Apart from HttpMethod.GET, we relate with delete

We copied the method body to our code.

```
1 usage new *
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback = restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
        = restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    ResponseEntity<FakeStoreProductDto> responseEntity = restTemplate.execute(specificProductURL, HttpMethod.DELETE,
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

Replaced this with restTemplate..

And related the get to delete.



```
ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor =
    restTemplate.responseEntityExtractor(FakeStoreProductDto.class);

    ResponseEntity<FakeStoreProductDto> responseEntity =
    restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
    responseExtractor, id);
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

ResponseExtractor will do basically, whatever response we get from rest template, it try to convert the response to the object , FakeStoreProductDto. When we make an HTTP call, it gives the reponse in JSON format, not in object format... we get the data in json, we need to extract the object out of JSON. This is what ResponseExtractor is doing.

Currently the restTemplate function has wrapped all the methods under some function... kind of FASADE all the method calls under its own wrapper. I don't bother about how rest Template were converting json obj... we are not bothered...

```
ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
```

Execute is the function where its making the HTTP call...

```
@Nullable
public <T> T execute(String uriTemplate, HttpMethod method, @Nullable
RequestCallback requestCallback, @Nullable ResponseExtractor<T>
responseExtractor, Object... uriVariables) throws RestClientException {
    URI url = this.getUriTemplateHandler().expand(uriTemplate, uriVariables);
    return this.doExecute(url, uriTemplate, method, requestCallback,
responseExtractor);
}
```

Its getting a java URL, getting a template handler.. expand and execute it.

Rest template is fasading lot of complexity form my side...

Delete product API:

```
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
=
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);

    ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

- This method is overriding some method inherited from a superclass or interface.
- It deletes a product from some data source based on the provided ID.
- It starts by creating a RestTemplate instance using restTemplateBuilder.

```
RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
```

Here, a RequestCallback is created. This callback is responsible for setting the appropriate accept headers for the HTTP request. It specifies that the response should be deserialized into a FakeStoreProductDto object.

```
ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor  
    = restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
```

A ResponseExtractor is created. This extractor is responsible for extracting the response entity from the HTTP response. In this case, it expects the response to be deserialized into a ResponseEntity containing a FakeStoreProductDto object.

```
ResponseEntity<FakeStoreProductDto> responseEntity =  
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,  
responseExtractor, id);
```

- The restTemplate.execute method is invoked. This method executes the HTTP DELETE request to the specificProductURL endpoint.
- It takes the URL, HTTP method (DELETE in this case), request callback, response extractor, and any additional parameters needed by the request (in this case, the product ID).
- It returns a ResponseEntity containing a FakeStoreProductDto object.

```
return getProductFromFakeStoreProductDto(responseEntity.getBody());
```

- Finally, the body of the ResponseEntity is extracted and passed to a method called getProductFromFakeStoreProductDto, which presumably converts the FakeStoreProductDto object into a Product object and returns it.

Overall, this code deletes a product by its ID using a RestTemplate instance, with custom request and response handling for serialization and deserialization of data.

Other than restTemplate many lib what handles http req..

RequestCall back: whenever u make a call via http, I need a place where I want my callback to comeback. A place response need to send back. Future-callbacks..

Read callback.. <https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/web/client/RequestCallback.html>

Package org.springframework.web.client

Interface RequestCallback

Functional Interface:

This is a functional interface and can therefore be used as the assignment target for a lambda expression or method reference.

@FunctionalInterface

public interface RequestCallback

Callback interface for code that operates on a [ClientHttpRequest](#). Allows manipulating the request headers, and write to the request body

Used internally by the [RestTemplate](#), but also useful for application code. There several available factory methods:

- [RestTemplate.acceptHeaderRequestCallback\(Class\)](#)
- [RestTemplate.httpEntityCallback\(Object\)](#)
- [RestTemplate.httpEntityCallback\(Object, Type\)](#)

what is acceptHeaderRequest... method doing exactly?

```
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
    =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
```

```
public <T> RequestCallback acceptHeaderRequestCallback(Class<T> responseType)
{
    return new AcceptHeaderRequestCallback(responseType);
}
```


```
public AcceptHeaderRequestCallback(@Nullable Type responseType) {
    this.responseType = responseType;
}
```

read about HEADER in HTTP call.....

when you expecting.. what kind of header which the service will send back. If more header then more validation..


when they send additional header.. from server.. we sent 6..







GET completed or • POST https://fakes • POST http://localh • **DEL http://localho** • + ▾ No E


http://localhost:8080/products/8 






DELETE ▾ **http://localhost:8080/products/8**

Params Auth **Headers (6)** Body Pre-req. Tests Settings

Headers  Hide auto-generated headers

	Key	Value	Descri...
<input checked="" type="checkbox"/>	Postman-Token	 <calculated when request is sent>	
<input checked="" type="checkbox"/>	Host	 <calculated when request is sent>	
<input checked="" type="checkbox"/>	User-Agent	 PostmanRuntime/7.36.3	
<input checked="" type="checkbox"/>	Accept	 */*	
<input checked="" type="checkbox"/>	Accept-Encoding	 gzip, deflate, br	
<input checked="" type="checkbox"/>	Connection	 keep-alive	
	Key	Value	Description

Body Cookies **Headers (5)** Test Results  200 OK 1415 ms 403

	Key	Value
	Content-Type	 application/json
	Transfer-Encoding	 chunked
	Date	 Sat, 16 Mar 2024 13:21:03 GMT
	Keep-Alive	 timeout=60
	Connection	 keep-alive

E can get 14 header. Custom header we can get.. tto handle those custom headers from server..

Here by default we are not sending any header.. we are sending query parameters.. request headers shown are default going with request..

Headers need separate class.

Currently responsibility of fakestore service is: CRUD operation, but its doing much more than that.. its interacting with 3rd party API: fakestore. Apart from that its doing something else...converting the object..

We have seen same thing in controller..

This Fakeprodservimpl doing many things. We should split it.... We can split 2 sections:

1. To handle 3rd party calls.
2. Do the mapping..

Out of these 2 the business logic: MAPPING.. handling 3rd party is not. Service should have mapping logic...

We will move handling 3 rd party.

AWS SDK..:

<https://aws.amazon.com/sdk-for-java/>

<https://docs.aws.amazon.com/sdk-for-java/latest/developer-guide/get-started.html>

```
package org.example;
```

```
import software.amazon.awssdk.services.s3.S3Client;
```

```
public class Handler {
    private final S3Client s3Client;

    public Handler() {
        s3Client = DependencyFactory.s3Client();
    }

    public void sendRequest() {
        // TODO: invoking the api calls using s3Client.
    }
}
```



```
createBucket(s3Client, bucket);

System.out.println("Uploading object...");

s3Client.putObject(PutObjectRequest.builder()
    .build(),
    RequestBody.fromString("Testing with"))

System.out.println("Upload complete");
System.out.printf("%n");

cleanUp(s3Client, bucket, key);

System.out.println("Closing the connection to");
s3Client.close();
System.out.println("Connection closed");
```

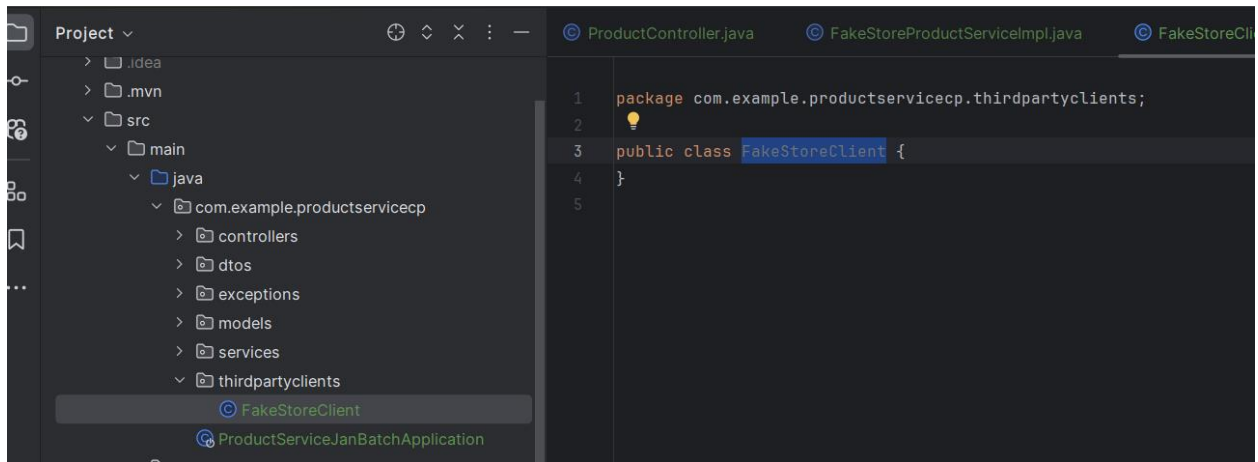
I am not worried how its calling S3. Or hwo http cpnnecting is getting created.

We will fasade everything rest template. And make the service clean..

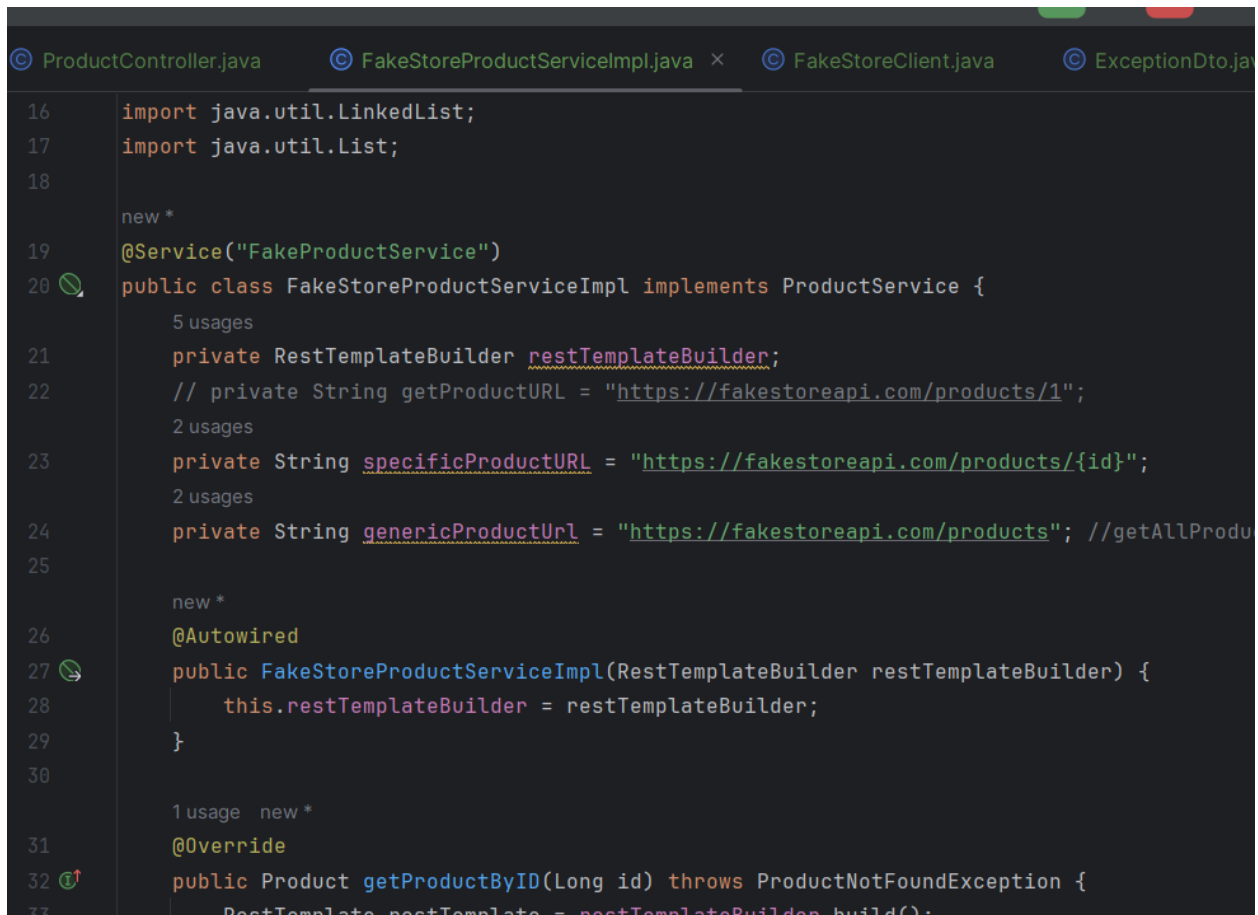
We put in client..

In majority of my use case I don't have to deal with restTeamplate, response entity, restcallback etc. will be provided by client integration.

We create a FakeStoreClient



It will have the logic from service.



We pasted everything from FakeStoreProductServiceImpl

```
@Service("FakeProductService")
public class FakeStoreProductServiceImpl implements ProductService {
    private RestTemplateBuilder restTemplateBuilder;
    // private String getProductURL = "https://fakestoreapi.com/products/1";
    private String specificProductURL =
"https://fakestoreapi.com/products/{id}";
    private String genericProductUrl = "https://fakestoreapi.com/products";
    //getAllProductUrl

    @Autowired
    public FakeStoreProductServiceImpl(RestTemplateBuilder
restTemplateBuilder) {
        this.restTemplateBuilder = restTemplateBuilder;
    }

    @Override
    public Product getProductByID(Long id) throws ProductNotFoundException {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.getForEntity(specificProductURL,
FakeStoreProductDto.class, id);
        if (responseEntity.getBody() == null) {
            // throw exception....
            throw new ProductNotFoundException("Product not found for id: " +
id);
        }
        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public List<Product> getAllProducts() {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto[]> responseEntity =
            restTemplate.getForEntity(genericProductUrl,
FakeStoreProductDto[].class);
        List<Product> productList = new LinkedList<>();
        for (FakeStoreProductDto fakeStoreProductDto :
responseEntity.getBody()) {
productList.add(getProductFromFakeStoreProductDto(fakeStoreProductDto));
        }
        return productList;
    }

    @Override
    public Product deleteProductById(Long id) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        // restTemplate.delete(specificProductURL, id);
        RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
        ResponseExtractor<ResponseEntity<FakeStoreProductDto>>
responseExtractor
            =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    }
```

```

        ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public Product addProduct(Product product) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.postForEntity(genericProductUrl,
getFakeStoreProductDtoFromProduct(product), FakeStoreProductDto.class);

        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public void updateProductById() {

    }

    //convert DTO to product
    private Product getProductFromFakeStoreProductDto(FakeStoreProductDto
fakeStoreProductDto) {
        Product product = new Product();
        product.setId(fakeStoreProductDto.getId());
        product.setTitle(fakeStoreProductDto.getTitle());
        product.setDescription(fakeStoreProductDto.getDescription());
        Category category = new Category();
        category.setName(fakeStoreProductDto.getCategory());
        product.setCategory(category);
        product.setPrice(fakeStoreProductDto.getPrice());

        return product;
    }

    //Reverse Mapper to convert product to dto.
    //I will get product from client.. but the fakestore API understands
fakestoreDTO..
    private FakeStoreProductDto getFakeStoreProductDtoFromProduct(Product
product) {
        FakeStoreProductDto fakeStoreProductDto = new FakeStoreProductDto();
        //we will not have the id from client..
        fakeStoreProductDto.setTitle(product.getTitle());
        fakeStoreProductDto.setDescription(product.getDescription());
        fakeStoreProductDto.setCategory(product.getCategory().getName());
        fakeStoreProductDto.setPrice(product.getPrice());

        return fakeStoreProductDto;
    }
}

```

Now changes in FakeStoreClient

1. We need a bean of FakestoreCLinet so we add @component:

When we are not sure if it's a service or controller we use @component. All helper goes as component

```
@Component
public class FakeStoreClient {
```

While it's true that `@Component` is a stereotype annotation in Spring that can be used to denote a bean, it's not specifically used when you're unsure whether a class should be a service or controller.

Here's the breakdown:

1. **@Component**: This annotation is a generic stereotype for any Spring-managed component. It indicates that a class is a Spring component and should be scanned by Spring's component scanner to create a bean. It's typically used for classes that don't fall into more specific categories like service, repository, or controller.
2. **@Service**: This annotation is used to denote a service component in the business layer of the application. It's a specialization of `@Component`, indicating that the annotated class is a service and is responsible for carrying out some business logic.
3. **@Controller**: This annotation is used to denote a controller component in the presentation layer of the application. It's also a specialization of `@Component`, indicating that the annotated class is a controller and handles incoming HTTP requests, typically by invoking service methods and preparing a response.

So, the decision between `@Component`, `@Service`, or `@Controller` depends on the role of the class within your application's architecture:

- Use `@Component` for general-purpose components that don't fit into specific layers.
- Use `@Service` for classes that implement business logic.
- Use `@Controller` for classes that handle HTTP requests and responses in a web application.

If you're unsure about the role of a class and you want it to be a Spring-managed component, you can start with `@Component` and later refactor it to `@Service` or `@Controller` as needed based on its responsibilities.

2. Change the fakestoreproductServiceImpl to fakestoreClient:

```
@Autowired
public FakeStoreClient(RestTemplateBuilder restTemplateBuilder) {
    this.restTemplateBuilder = restTemplateBuilder;
}
```

3. Also `FakeStoreClient` is not implementing so we will remove the override.. from `getProductById` same for `getAllProducts`.

```
no usages new *
31 @Override
32 public Product getProductById(Long id) throws ProductNotFoundException {
33     RestTemplate restTemplate = restTemplateBuilder.build();
34     ResponseEntity<FakeStoreProductDto> responseEntity
35         = restTemplate.getForEntity(specificProductURL, FakeStoreProductDto.class);
36     if (responseEntity.getBody() == null) {
37         // throw exception
38     }
39 }

no usages new *
43 @Override
44 public List<Product> getAllProducts() {
45     RestTemplate restTemplate = restTemplateBuilder.build();
46     ResponseEntity<FakeStoreProductDto[]> responseEntity =
47         restTemplate.getForEntity(genericProductUrl, FakeStoreProductDto[].class);
48     List<Product> productList = new LinkedList<>();
49     for (FakeStoreProductDto fakeStoreProductDto : responseEntity.getBody()) {
50         Product product = new Product();
51         product.setId(fakeStoreProductDto.getId());
52         product.setName(fakeStoreProductDto.getName());
53         product.setDescription(fakeStoreProductDto.getDescription());
54         product.setPrice(fakeStoreProductDto.getPrice());
55         product.setCategory(fakeStoreProductDto.getCategory());
56         productList.add(product);
57     }
58     return productList;
59 }
```

Same for delete: `deleteProductById` and add a product method.

```
no usages new *
54
55 @Override
56 public Product deleteProductById(Long id) {
57     RestTemplate restTemplate = restTemplateBuilder.build();
58     // restTemplate.delete(specificProductURL, id);
59     RequestCallback requestCallback = restTemplate.acceptHeaderRequestCallback(Product.class);
60     ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor =
61         restTemplate.getResponseExtractor(FakeStoreProductDto.class);
62     ResponseEntity<FakeStoreProductDto> responseEntity = restTemplate.exchange(
63         specificProductURL, HttpMethod.DELETE, requestCallback, responseExtractor, id);
64     return responseEntity.getBody();
65 }
```

Same way from all..

The FakestoreCLinet is a client for FakeStore.

```
public Product getProductById(Long id) throws ProductNotFoundException {
```

The function getProductById should return a fakeStoreProductDto or a product?

They don't need to be worry about product... in all the method we will return the DTO not the product.

So we don't need the methods getProductFromFakeStoreProductDto and
getFakeStoreProductDtoFromProduct

For example in addAproduct we changed the product into dto..

```
public FakeStoreProductDto addProduct(FakeStoreProductDto  
fakeStoreProductDto) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
    ResponseEntity<FakeStoreProductDto> responseEntity  
        = restTemplate.postForEntity(genericProductUrl,  
getFakeStoreProductDtoFromProduct(fakeStoreProductDto),  
FakeStoreProductDto.class);  
  
    return getProductFromFakeStoreProductDto(responseEntity.getBody());  
}
```

we will remove the mapper logic from here. Whatever we are getting as a response, we are returning the same thing. Removed all conversion logic..

```
public FakeStoreProductDto addProduct(FakeStoreProductDto  
fakeStoreProductDto) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
    ResponseEntity<FakeStoreProductDto> responseEntity  
        = restTemplate.postForEntity(genericProductUrl,  
fakeStoreProductDto, FakeStoreProductDto.class);  
  
    return responseEntity.getBody();  
}
```

same way we will remove the conversion logic from delete a product

```
public FakeStoreProductDto deleteProductById(Long id) {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
    // restTemplate.delete(specificProductURL, id);  
    RequestCallback requestCallback =  
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);  
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor  
        =  
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
```

```

        ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

```

```

        return responseEntity.getBody();
    }

```

GetAllProduct:

```

public List<FakeStoreProductDto> getAllProducts() {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto[]> responseEntity =
        restTemplate.getForEntity(genericProductUrl,
FakeStoreProductDto[].class);
    List<Product> productList = new LinkedList<>();
    for (FakeStoreProductDto fakeStoreProductDto : responseEntity.getBody())
    {
        productList.add(fakeStoreProductDto);
    }
    return productList;
}

```

changed to

```

public FakeStoreProductDto[] getAllProducts() {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto[]> responseEntity =
        restTemplate.getForEntity(genericProductUrl,
FakeStoreProductDto[].class);

    return responseEntity.getBody();
}

```

1. Method Signature Change:

- You changed the return type of the method from `List<FakeStoreProductDto>` to `FakeStoreProductDto[]`. This means instead of returning a list of products, it now returns an array of products.

2. RestTemplate Usage:

- You're still using RestTemplate to make a GET request to the `genericProductUrl`.
- The response is retrieved as a ResponseEntity containing an array of `FakeStoreProductDto`.

3. Return Statement:

- You're returning the body of the `ResponseEntity` directly, which is the array of `FakeStoreProductDto`.

This change simplifies the method by directly returning the array of products received from the API response without iterating over it to convert it into a list. It's a more concise approach if you only need to work with the array of products without requiring it to be in a list format.

Your code change seems straightforward. Let's break it down:

```
java Copy code

public FakeStoreProductDto[] getAllProducts() {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto[]> responseEntity =
        restTemplate.getForEntity(genericProductUrl, FakeStoreProductDto[].class);

    return responseEntity.getBody();
}
```



ChatGPT

`responseEntity.getBody()` is a method call that retrieves the body of the HTTP response encapsulated by the `ResponseEntity` object.

In the context of the provided code snippet:

```
java Copy code

ResponseEntity<FakeStoreProductDto[]> responseEntity =
    restTemplate.getForEntity(genericProductUrl, FakeStoreProductDto[].class);
```

Here, `responseEntity` represents the HTTP response received from the server after making a GET request to the `genericProductUrl` URL. The response body contains an array of `FakeStoreProductDto` objects.

`getBody()` is a method provided by the `ResponseEntity` class. It returns the body of the HTTP response. In this case, since the response body is expected to be an array of `FakeStoreProductDto` objects, the return type of `getBody()` is `FakeStoreProductDto[]`.

So, `responseEntity.getBody()` retrieves the array of `FakeStoreProductDto` objects from the HTTP response, allowing you to access and work with the actual data returned by the server.



GetProductByID:

```
public FakeStoreProductDto getProductByID(Long id) throws
ProductNotFoundException {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto> responseEntity
        = restTemplate.getForEntity(specificProductURL,
FakeStoreProductDto.class, id);
    if (responseEntity.getBody() == null) {
        // throw exception...
        throw new ProductNotFoundException("Product not found for id: " +
id);
    }
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

so now the class FakeStoreClient is fixed now. No error:

```
@Component
public class FakeStoreClient {
    private RestTemplateBuilder restTemplateBuilder;
    // private String getProductURL = "https://fakestoreapi.com/products/1";
    private String specificProductURL =
"https://fakestoreapi.com/products/{id}";
    private String genericProductUrl = "https://fakestoreapi.com/products";
    //getAllProductUrl

    @Autowired
    public FakeStoreClient(RestTemplateBuilder restTemplateBuilder) {
        this.restTemplateBuilder = restTemplateBuilder;
    }

    public FakeStoreProductDto getProductByID(Long id) throws
ProductNotFoundException {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.getForEntity(specificProductURL,
FakeStoreProductDto.class, id);
        if (responseEntity.getBody() == null) {
            // throw exception...
            throw new ProductNotFoundException("Product not found for id: " +
id);
        }
        return responseEntity.getBody();
    }

    public FakeStoreProductDto[] getAllProducts() {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto[]> responseEntity =
            restTemplate.getForEntity(genericProductUrl,
FakeStoreProductDto[].class);

        return responseEntity.getBody();
    }
}
```

```

    }
    /*
    responseEntity represents the HTTP response received from the server after
    making a GET request to the genericProductUrl URL. The response body contains
    an array of FakeStoreProductDto objects.
    getBody() is a method provided by the ResponseEntity class. It returns the
    body of the HTTP response. In this case, since the response body is expected
    to be an array of FakeStoreProductDto objects, the return type of getBody()
    is FakeStoreProductDto[].
    */
    public FakeStoreProductDto deleteProductById(Long id) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        // restTemplate.delete(specificProductUrl, id);
        RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
        ResponseExtractor<ResponseEntity<FakeStoreProductDto>>
responseExtractor
        =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);

        ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductUrl, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
        return responseEntity.getBody();
    }

    public FakeStoreProductDto addProduct(FakeStoreProductDto
fakeStoreProductDto) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.postForEntity(genericProductUrl,
fakeStoreProductDto, FakeStoreProductDto.class);

        return responseEntity.getBody();
    }

    public void updateProductById() {
    }
}

```

Now we will fix the fakestoreService:

```
@Service("FakeProductService")
public class FakeStoreProductServiceImpl implements ProductService {
    private RestTemplateBuilder restTemplateBuilder;
    // private String getProductURL = "https://fakestoreapi.com/products/1";
    private String specificProductURL =
"https://fakestoreapi.com/products/{id}";
    private String genericProductUrl = "https://fakestoreapi.com/products";
    //getAllProductUrl

    @Autowired
    public FakeStoreProductServiceImpl(RestTemplateBuilder
restTemplateBuilder) {
        this.restTemplateBuilder = restTemplateBuilder;
    }

    @Override
    public Product getProductById(Long id) throws ProductNotFoundException {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.getForEntity(specificProductURL,
FakeStoreProductDto.class, id);
        if (responseEntity.getBody() == null) {
            // throw exception....
            throw new ProductNotFoundException("Product not found for id: " +
id);
        }
        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public List<Product> getAllProducts() {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto[]> responseEntity =
            restTemplate.getForEntity(genericProductUrl,
FakeStoreProductDto[].class);
        List<Product> productList = new LinkedList<>();
        for (FakeStoreProductDto fakeStoreProductDto :
responseEntity.getBody()) {
            productList.add(getProductFromFakeStoreProductDto(fakeStoreProductDto));
        }
        return productList;
    }

    @Override
    public Product deleteProductById(Long id) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        // restTemplate.delete(specificProductURL, id);
        RequestCallback requestCallback =
restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
        ResponseExtractor<ResponseEntity<FakeStoreProductDto>>
responseExtractor
            =
restTemplate.responseEntityExtractor(FakeStoreProductDto.class);
    }
```

```

        ResponseEntity<FakeStoreProductDto> responseEntity =
restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
responseExtractor, id);
        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public Product addProduct(Product product) {
        RestTemplate restTemplate = restTemplateBuilder.build();
        ResponseEntity<FakeStoreProductDto> responseEntity
            = restTemplate.postForEntity(genericProductUrl,
getFakeStoreProductDtoFromProduct(product), FakeStoreProductDto.class);

        return getProductFromFakeStoreProductDto(responseEntity.getBody());
    }

    @Override
    public void updateProductById() {

    }

    //convert DTO to product
    private Product getProductFromFakeStoreProductDto(FakeStoreProductDto
fakeStoreProductDto) {
        Product product = new Product();
        product.setId(fakeStoreProductDto.getId());
        product.setTitle(fakeStoreProductDto.getTitle());
        product.setDescription(fakeStoreProductDto.getDescription());
        Category category = new Category();
        category.setName(fakeStoreProductDto.getCategory());
        product.setCategory(category);
        product.setPrice(fakeStoreProductDto.getPrice());

        return product;
    }

    //Reverse Mapper to convert product to dto.
    //I will get product from client.. but the fakestore API understands
fakestoreDTO..
    private FakeStoreProductDto getFakeStoreProductDtoFromProduct(Product
product) {
        FakeStoreProductDto fakeStoreProductDto = new FakeStoreProductDto();
        //we will not have the id from client..
        fakeStoreProductDto.setTitle(product.getTitle());
        fakeStoreProductDto.setDescription(product.getDescription());
        fakeStoreProductDto.setCategory(product.getCategory().getName());
        fakeStoreProductDto.setPrice(product.getPrice());

        return fakeStoreProductDto;
    }

    /*
map<String, Object>
    */
}

```

The service now should be depend on FakeStoreClient instead of the below:

```
private RestTemplateBuilder restTemplateBuilder;  
// private String getProductURL = "https://fakestoreapi.com/products/1";  
private String specificProductURL = "https://fakestoreapi.com/products/{id}";  
private String genericProductUrl = "https://fakestoreapi.com/products";  
//getAllProductUrl
```

```
@Service("FakeProductService")  
public class FakeStoreProductServiceImpl implements ProductService {  
  
    private FakeStoreClient fakeStoreClient;
```

This should depend on fakestoreclient:

```
@Autowired  
public FakeStoreProductServiceImpl(RestTemplateBuilder restTemplateBuilder) {  
    this.restTemplateBuilder = restTemplateBuilder;  
}
```

```
private FakeStoreClient fakeStoreClient;  
@Autowired  
public FakeStoreProductServiceImpl(FakeStoreClient fakeStoreClient) {  
    this.fakeStoreClient = fakeStoreClient;  
}
```

I can remove all logic from getProductById:

```
@Override  
public Product getProductByID(Long id) throws ProductNotFoundException {  
    RestTemplate restTemplate = restTemplateBuilder.build();  
    ResponseEntity<FakeStoreProductDto> responseEntity  
        = restTemplate.getForEntity(specificProductURL,  
FakeStoreProductDto.class, id);  
    if (responseEntity.getBody() == null) {  
        // throw exception....  
        throw new ProductNotFoundException("Product not found for id: " +  
id);  
    }  
    return getProductFromFakeStoreProductDto(responseEntity.getBody());  
}
```

Changed to

```
@Override  
public Product getProductByID(Long id) throws ProductNotFoundException {  
    return  
getProductFromFakeStoreProductDto(fakeStoreClient.getProductById(id));  
}
```

Same way remove all logic from `getAllProducts`:

```
@Override
public List<Product> getAllProducts() {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto[]> responseEntity =
        restTemplate.getForEntity(genericProductUrl,
            FakeStoreProductDto[].class);
    List<Product> productList = new LinkedList<>();
    for (FakeStoreProductDto fakeStoreProductDto : responseEntity.getBody())
    {

        productList.add(getProductFromFakeStoreProductDto(fakeStoreProductDto));
    }
    return productList;
}
```

Changed to:

```
@Override
public List<Product> getAllProducts() {
    List<Product> productList = new LinkedList<>();
    for (FakeStoreProductDto fakeStoreProductDto :
        fakeStoreClient.getAllProducts()) {

        productList.add(getProductFromFakeStoreProductDto(fakeStoreProductDto));
    }
    return productList;
}
```

Same way remove all logic from `DeleteProductById`:

```
@Override
public Product deleteProductById(Long id) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    // restTemplate.delete(specificProductURL, id);
    RequestCallback requestCallback =
        restTemplate.acceptHeaderRequestCallback(FakeStoreProductDto.class);
    ResponseExtractor<ResponseEntity<FakeStoreProductDto>> responseExtractor
        =
        restTemplate.responseEntityExtractor(FakeStoreProductDto.class);

    ResponseEntity<FakeStoreProductDto> responseEntity =
        restTemplate.execute(specificProductURL, HttpMethod.DELETE, requestCallback,
            responseExtractor, id);
    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

To:

```
@Override
public Product deleteProductById(Long id) {
    return
        getProductFromFakeStoreProductDto(fakeStoreClient.deleteProductById(id));
}
```

Same way adda Product:

```
@Override
public Product addProduct(Product product) {
    RestTemplate restTemplate = restTemplateBuilder.build();
    ResponseEntity<FakeStoreProductDto> responseEntity
        = restTemplate.postForEntity(genericProductUrl,
getFakeStoreProductDtoFromProduct(product), FakeStoreProductDto.class);

    return getProductFromFakeStoreProductDto(responseEntity.getBody());
}
```

Change to:

```
@Override
public Product addProduct(Product product) {
    return getProductFromFakeStoreProductDto(
fakeStoreClient.addProduct(getFakeStoreProductDtoFromProduct(product)));
}
```

Now overall service looks cleaner. We had refactor all in beautiful code. Everything is wrapped in client. Tomorrow if client changes, URL, Format.. anything, our service will not be impacted...

Max impact we get is some of the mapping logic will change. As long as contract remain same we don't have any impact. We can change from one fakestore to another only thing changes is mapping logic. This is an variation of ADAPTER design pattern.

```
@Service("FakeProductService")
public class FakeStoreProductServiceImpl implements ProductService {

    private FakeStoreClient fakeStoreClient;

    @Autowired
    public FakeStoreProductServiceImpl(FakeStoreClient fakeStoreClient) {
        this.fakeStoreClient = fakeStoreClient;
    }

    @Override
    public Product getProductById(Long id) throws ProductNotFoundException {
        return
getProductFromFakeStoreProductDto(fakeStoreClient.getProductById(id));
    }

    @Override
    public List<Product> getAllProducts() {
        List<Product> productList = new LinkedList<>();
        for (FakeStoreProductDto fakeStoreProductDto :
fakeStoreClient.getAllProducts()) {
productList.add(getProductFromFakeStoreProductDto(fakeStoreProductDto));
        }
    }
}
```



```

        return productList;
    }

    @Override
    public Product deleteProductById(Long id) {
        return
getProductFromFakeStoreProductDto(fakeStoreClient.deleteProductById(id));
    }

    @Override
    public Product addProduct(Product product) {
        return getProductFromFakeStoreProductDto(
fakeStoreClient.addProduct(getFakeStoreProductDtoFromProduct(product)));
    }

    @Override
    public void updateProductById() {

    }

    //convert DTO to product
    private Product getProductFromFakeStoreProductDto(FakeStoreProductDto
fakeStoreProductDto) {
        Product product = new Product();
        product.setId(fakeStoreProductDto.getId());
        product.setTitle(fakeStoreProductDto.getTitle());
        product.setDescription(fakeStoreProductDto.getDescription());
        Category category = new Category();
        category.setName(fakeStoreProductDto.getCategory());
        product.setCategory(category);
        product.setPrice(fakeStoreProductDto.getPrice());

        return product;
    }

    //Reverse Mapper to convert product to dto.
    //I will get product from client.. but the fakestore API understands
fakestoreDTO..
    private FakeStoreProductDto getFakeStoreProductDtoFromProduct(Product
product) {
        FakeStoreProductDto fakeStoreProductDto = new FakeStoreProductDto();
        //we will not have the id from client..
        fakeStoreProductDto.setTitle(product.getTitle());
        fakeStoreProductDto.setDescription(product.getDescription());
        fakeStoreProductDto.setCategory(product.getCategory().getName());
        fakeStoreProductDto.setPrice(product.getPrice());

        return fakeStoreProductDto;
    }

```

Why didn't we create an Interface:

There is a fakestore, tomorrow can be dummy store. Not surely the dummystore client and dummystore productDtop will not be same as fakestore..

if not then in the FakeStoreProductServiceImpl for dummy store service or we should create a new service. Or repurpose the fakestore... always create new service..

if we are creating a new service For a new service.. for a new 3rd party abstraction is happening at service layer... FakeStoreProductServiceImpl

The client FakeStoreClient – we are creating or always come from 3rd Party... here just to show how 3rd party client are created. In reality client will come via a library. And we don't have access to the code..

fakeStoreClient this will be a interface which they will implement. They can decide if they want it as a interface or a direct implementation...

generally it will not be interface. there will be single way of client creation...

as not 2 3rd party will send us in similar format. The dummy n fakestore will not send same kind of client...

we could use generics? U can, but overkill. If you are not writing a library...

is the package name Client industry standard ? yes.

decision making is at controller level which service to use ? yes at controller level. Or generally you do via configuration, which client you want use. Generally there will be flags.

can u please explain multiple getters used for addProduct in impl class? Last class.

Can we use multiple clients? Yes. For that create multiple services. Then in controller decide which service to be used.. here now we are deciding via qualifier:

```
//@Controller
@RestController
@RequestMapping("/products")
public class ProductController {
    private ProductService productService;
    @Autowired
    public ProductController(@Qualifier("FakeProductService") ProductService
productService) {
        this.productService = productService;
    }
}
```

Explanation and flow of the two important classes: FakeStoreProductServiceImpl, FakeStoreClient

The provided code represents a simplified implementation of a client for interacting with a fake store API. Let's break down the request flow through this client:

1. Instantiation and Configuration:

- A **FakeStoreClient** bean is instantiated as a Spring component (**@Component**). It is configured with a **RestTemplateBuilder** injected via constructor injection. The **RestTemplateBuilder** is used to create instances of **RestTemplate**, a class provided by Spring for making HTTP requests.

2. API Endpoints:

- The client defines two endpoints:
 - specificProductURL**: Endpoint for retrieving a specific product by its ID.
 - genericProductUrl**: Endpoint for retrieving all products.

3. Client Methods:

- The client provides several methods for interacting with the fake store API:
 - getProductById(Long id)**: Retrieves a product by its ID.
 - getAllProducts()**: Retrieves all products.
 - deleteProductById(Long id)**: Deletes a product by its ID.
 - addProduct(FakeStoreProductDto fakeStoreProductDto)**: Adds a new product.
 - updateProductById()**: Placeholder for updating a product (not implemented).

4. Service Layer:

- A service class (**FakeStoreProductServiceImpl**) is defined, implementing the **ProductService** interface.
- This service class delegates product-related operations to the **FakeStoreClient**.

5. Service Methods:

- The service class provides methods for retrieving, deleting, adding, and updating products, which internally call the respective methods of the **FakeStoreClient**.
- These methods handle any necessary conversions between DTOs (Data Transfer Objects) used by the client and domain objects used internally.

6. Request Flow:

- When a request is made to retrieve, delete, add, or update a product:
 - The service method delegates the request to the corresponding method of the **FakeStoreClient**.
 - The **FakeStoreClient** uses **RestTemplate** to make HTTP requests to the fake store API endpoints.
 - The response received from the API is processed and possibly converted between DTOs and domain objects.
 - Finally, the result is returned to the caller.

In summary, the request flow involves client methods invoking **RestTemplate** to interact with the fake store API, with service methods acting as intermediaries for handling business logic and conversions between DTOs and domain objects.

The decision to create a separate class like `FakeStoreClient` instead of directly writing the logic inside the service (`FakeStoreProductServiceImpl`) can be based on several factors:

1. **Separation of Concerns (SoC):**

- By creating a separate client class, you can isolate the code responsible for communicating with the external API from the service layer. This adheres to the principle of separation of concerns, making the codebase easier to understand, maintain, and test.

2. **Reusability:**

- Separating the API communication logic into its own class allows you to reuse this functionality across multiple services or even in different parts of the application. It promotes code reusability and avoids duplication.

3. **Testability:**

- A separate client class can be easily mocked or stubbed in unit tests, allowing you to test the service layer in isolation without needing to make actual HTTP requests to the external API. This improves testability and makes unit testing more efficient.

4. **Flexibility:**

- Separating the client logic allows you to easily switch to a different API or change the implementation details without affecting the service layer. It provides flexibility in adapting to changes in the external API or integrating with different APIs in the future.

5. **Code Organization:**

- Keeping the code organized and modular is another benefit. Having a dedicated client class with methods specifically for interacting with the external API makes the codebase more structured and easier to navigate.

Overall, creating a separate client class offers several advantages in terms of maintainability, reusability, testability, flexibility, and code organization, which may justify the decision to abstract away the API communication logic from the service layer.

ProductControllerAdvice explanation:

```
@RestControllerAdvice(assignableTypes = ProductController.class)
// @RestControllerAdvice: This annotation is a combination of
// @ControllerAdvice and @ResponseBody. It allows you to use this class to
// handle exceptions thrown from controllers and automatically convert the
// response to JSON using the ResponseEntity class.
public class ProductControllerAdvice {
    // Exception handling method
    @ExceptionHandler(ProductNotFoundException.class)
    // @ResponseStatus(HttpStatus.NOT_FOUND)
    // @ResponseBody
    private ResponseEntity<ExceptionDto>
handleProductNotFoundException(ProductNotFoundException e){
    ExceptionDto exceptionDto = new ExceptionDto();
    exceptionDto.setMessage(e.getMessage());
    exceptionDto.setStatus("failure");
    ResponseEntity<ExceptionDto> responseEntity
        = new ResponseEntity<>(exceptionDto, HttpStatus.NOT_FOUND);

    return responseEntity;
}
}
```

The provided code is an example of an exception handling advice in a Spring Boot application, specifically for the `ProductController` class.

Let's break down the purpose and functionality of this code:

1. **Annotations:**

- **@RestControllerAdvice:** This annotation is a combination of **@ControllerAdvice** and **@ResponseBody**. It allows you to use this class to handle exceptions thrown from controllers and automatically convert the response to JSON using the **ResponseEntity** class.

2. **Exception Handling Method:**

- **@ExceptionHandler(ProductNotFoundException.class):** This annotation specifies that the following method will handle exceptions of type **ProductNotFoundException**.
- **private ResponseEntity<ExceptionDto> handleProductNotFoundException(ProductNotFoundException e):** This method takes a **ProductNotFoundException** as a parameter and returns a **ResponseEntity<ExceptionDto>**.
- Inside the method, an **ExceptionDto** object is created to encapsulate information about the exception. In this case, it includes the exception message and a status indicating failure.
- Then, a **ResponseEntity** is created with the **ExceptionDto** object and a status of **NOT_FOUND**.
- Finally, the **ResponseEntity** is returned.

3. **Purpose:**

- The purpose of this advice class is to provide centralized exception handling for the **ProductController**. When a **ProductNotFoundException** occurs within any method of the

ProductController, Spring will intercept the exception and invoke the **handleProductNotFoundException** method in this advice class.

- The method then constructs an appropriate response entity (**ExceptionDto** wrapped in **ResponseEntity**) with the details of the exception and returns it to the client.
- This approach helps in maintaining a consistent error response format across the application and separates the error handling logic from the controller methods, improving code readability and maintainability.

Overall, this advice class ensures that exceptions thrown by the **ProductController** are properly handled and transformed into meaningful error responses for the client.

In the service FakeStoreProductServiceImpl .. fakeStoreClient it is setting outside my service, if it's 3rd party system and not a IF I cant do anything to achieve DI.

Kumar: We will see how Razorpay api is integrated.

99% time we use the client given by 3rd party api. No need to use rest template.

Affan: If I were to work here. Since I integrate a 3rd party API. I will create a interface with generic as data type would be different. Logic will be different. In IF in each method I will encapsulate the logic of that 3rd party service.

Nikhil: if I have multiple 3rd party and I want to generalize. Not every 3rd party will follow the standard. I have to handle all 3rd party. I will have to do the overhead to make sure that 3rd party response coming in. In the generics which is getting handled so that out of generics I am getting the similar response... I have to handle every 3rd party separately in multiple functions in the generics.. so that when I send out response from generics it is similar response.

Affan: when you are creating the FakeStoreProductServiceImpl , it would be initialize as ... lets say I have a storeClient IF and within that storeClient generic would be FakeStoreProductDTO. But in generic I will have to handle DTO separately so that it sends out the fakestore product DTO.

Instead of that if I create multiple service which is simple implementation and less risky. A good article about abstraction. If u overuse abstraction... problem happens. Use abstraction till it helps you.

From the conversation between Nikhil and Affan, it seems they are discussing different approaches to handling interactions with third-party APIs, particularly in the context of the `FakeStoreClient` class and its integration within the `FakeStoreProductServiceImpl`.

Let's break down the key points and discuss how generics could be used:

1. Nikhil's Approach:

Nikhil suggests primarily using the `FakeStoreClient` provided by the third-party API. This implies directly integrating the specific client provided by the API vendor (`FakeStoreClient` in this case) into the application logic.

2. Affan's Approach:

Affan proposes a more abstract and flexible approach. He suggests creating an interface that encapsulates the logic for interacting with third-party APIs. Each method in the interface would handle the specifics of communicating with the respective third-party service. This approach allows for better encapsulation and potentially easier maintenance if there are multiple third-party APIs involved.

3. Generalization for Multiple Third-Party APIs:

Nikhil mentions a scenario where there might be multiple third-party APIs, each with its own nuances and response formats. In such cases, he suggests using generics to create a more generalized solution. The idea is to have a generic interface that can handle interactions with various third-party APIs, with each implementation tailored to the specific API.

The challenge here is to design the generics in such a way that they can accommodate the differences between the APIs while still providing a consistent interface to the application. This might involve handling different DTOs, response formats, error handling mechanisms, etc., within the generic implementation.

```
java
// Generic interface for interacting with third-party APIs
public interface ThirdPartyApiClient<T> {
    T getProductById(Long id) throws ProductNotFoundException;
    List<T> getAllProducts();
    T deleteProductById(Long id);
    T addProduct(T product);
    void updateProductById();
}
```

```

// Implementation for the FakeStore API
@Component
public class FakeStoreClient implements ThirdPartyApiClient<FakeStoreProductDto> {
    // Implementation details specific to the FakeStore API
    // ...
}

// Service implementation using generics
public class ProductServiceWithGenerics<T> {
    private ThirdPartyApiClient<T> thirdPartyApiClient;

    @Autowired
    public ProductServiceWithGenerics(ThirdPartyApiClient<T> thirdPartyApiClient) {
        this.thirdPartyApiClient = thirdPartyApiClient;
    }

    // Service methods can now operate with any type of third-party API client
    public T getProductById(Long id) throws ProductNotFoundException {
        return thirdPartyApiClient.getProductById(id);
    }

    // Other methods omitted for brevity
    // ...
}

```

In this setup:

- The `ThirdPartyApiClient` interface is generic, allowing it to work with any DTO type representing products from different third-party APIs.
- Each implementation of the `ThirdPartyApiClient` interface corresponds to a specific third-party API (e.g., `FakeStoreClient`).
- The `ProductServiceWithGenerics` class can use the generic `ThirdPartyApiClient` interface to interact with any third-party API, providing flexibility and scalability.

This approach enables easier integration with multiple third-party APIs while maintaining a consistent interface within the application. However, handling the differences between various APIs still requires careful consideration and potentially some level of customization within each implementation of the `ThirdPartyApiClient` interface.

Certainly! Let's break down the flow of the system using the provided code and the proposed approach with generics.

1. Initialization:

- In the application context, Spring Boot initializes beans, including the `FakeStoreClient` and the `ProductServiceWithGenerics`.
- The `FakeStoreClient` bean is instantiated with `RestTemplateBuilder` injected, allowing it to make HTTP requests to the Fake Store API.
- The `ProductServiceWithGenerics` bean is also instantiated, with the `FakeStoreClient` passed as a dependency.

2. Request Handling:

- When a request comes in to retrieve, delete, add, or update a product, it is routed to the corresponding method in `ProductServiceWithGenerics`.
- For example, if a request is made to retrieve a product by ID, `getProductById()` method in `ProductServiceWithGenerics` is invoked.

3. Invocation of Third-Party API Client:

- Inside `ProductServiceWithGenerics`, the method delegates the request to the corresponding method of the injected `ThirdPartyApiClient`, which in this case is the `FakeStoreClient`.
- For instance, in the `getProductById()` method, it calls `getProductById()` method of the `FakeStoreClient`.

4. Communication with Fake Store API:

- The `FakeStoreClient` interacts with the Fake Store API using `RestTemplate`.
- For example, in `getProductById()` method of `FakeStoreClient`, it makes an HTTP GET request to the Fake Store API endpoint to retrieve the product by ID.

5. Processing of Response:

- The `FakeStoreClient` receives a response from the Fake Store API.
- The response might be in the form of DTOs specific to the Fake Store API.

6. Conversion (if necessary):

- If needed, the response DTOs from the Fake Store API are converted to domain objects. This conversion can be handled within the `FakeStoreClient` or `ProductServiceWithGenerics` using mapping methods.

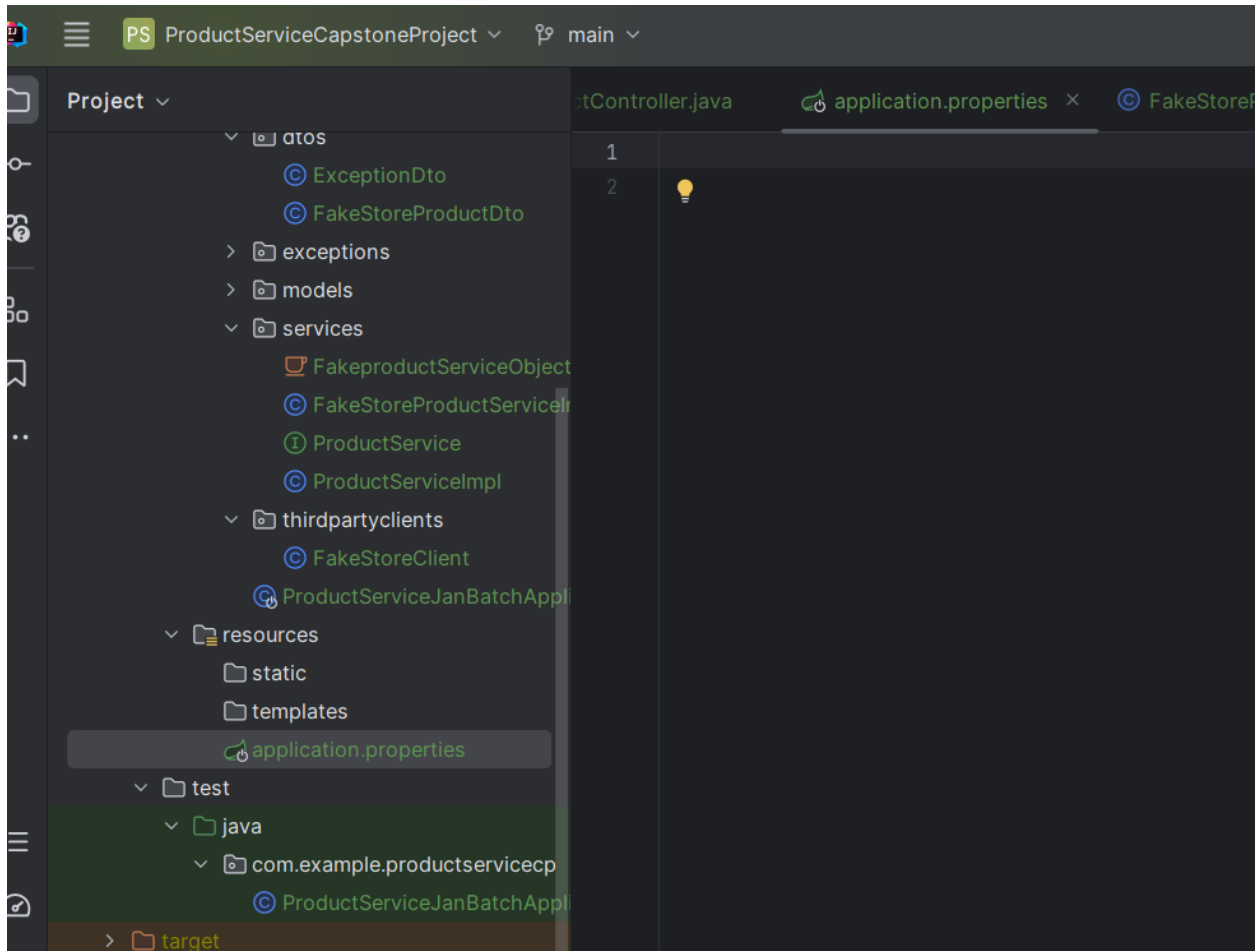
7. Returning Response:

- The converted domain objects or DTOs are returned from the `getProductById()` method of `ProductServiceWithGenerics`.
- Finally, the response is returned to the caller, completing the request handling process.

Overall, this flow ensures that the application can interact with the Fake Store API (or potentially other third-party APIs) in a flexible and maintainable way. The use of generics allows for a standardized interface (`ThirdPartyApiClient`) to interact with different APIs, while still accommodating the specific details and response formats of each API through individual implementations (`FakeStoreClient`, for example).

After Break:

Application.ppty:



What all we should be adding here. It's nothing but a configuration file.. which contains key: value pair..

Any key and any values... what all we should put here..

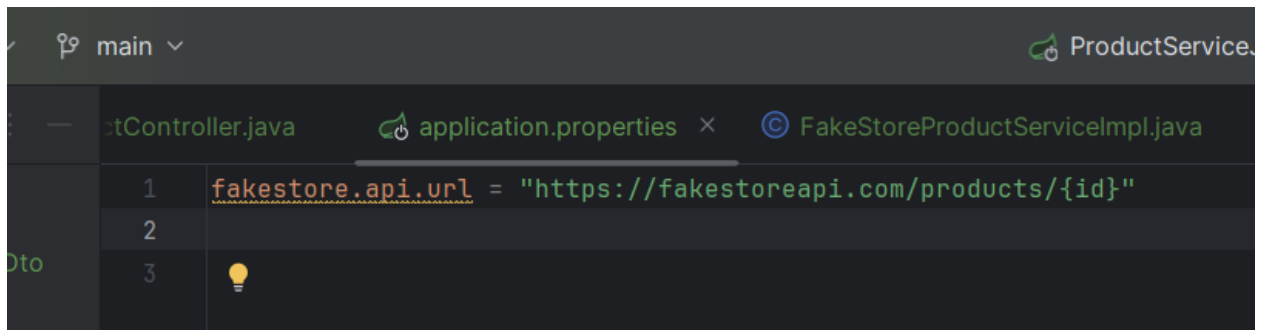
```
@Component
public class FakeStoreClient {
    private RestTemplateBuilder restTemplateBuilder; // RestTemplateBuilder
    // is used to create instances of RestTemplate
    // retrieving a specific product by its ID: specificProductURL and
    // genericProductUrl for all products
    private String specificProductURL =
    "https://fakestoreapi.com/products/{id}";
    private String genericProductUrl = "https://fakestoreapi.com/products";
    //getAllProductUrl
}
```

This specificURL should not be hardcoded in code... this is a classic example. These URL should go in app.properties..

Before:

```
// the client provides several methods for interacting with the fake store
API: getProductById, getAllProducts, deleteProductById, addProduct and
updateProductById.
@Autowired
public FakeStoreClient(RestTemplateBuilder restTemplateBuilder) {
    this.restTemplateBuilder = restTemplateBuilder;
}
```

After:



Should be without quotes:

```
fakestore.api.url = https://fakestoreapi.com/products/{id}
```

```
@Autowired
public FakeStoreClient(RestTemplateBuilder restTemplateBuilder,
    @Value("${fakestore.api.url}") String fakeStoreUrl) {
    this.restTemplateBuilder = restTemplateBuilder;
    this.specificProductURL = fakeStoreUrl;
}
```

@value: allow us to read an element from application.properties. we want to read the element fakestore.api.url.

We need to add it via constructor.

Adv: this property can be used in multiple places. Also I have to change once in properties file.

We can make it work via field injection as well.

```

23 private RestTemplateBuilder restTemplateBuilder;
24
25 @Value("${fakestore.api.url}")
26 private String specificProductUrl;
27
28 private String genericProductUrl = "https://fakestoreapi.com/products";
29
30 @Autowired
31 public FakeStoreClient(RestTemplateBuilder restTemplateBuilder) {
32     this.restTemplateBuilder = restTemplateBuilder;
33     //this.specificProductUrl = fakestoreurl;
34 }

```

This is similar to field injection. We should avoid it.

```

public class FakeStoreClient {

    5 usages
    private RestTemplateBuilder restTemplateBuilder;

    @Value("${fakestore.api.url}")
    private String specificProductUrl;
    2 usages
    private String genericProductUrl = "https://fakestoreapi.com/products";

    @Autowired
    public FakeStoreClient(RestTemplateBuilder restTemplateBuilder){
        // @Value("${fakestore.api.url}") String fakestoreurl) {
        this.restTemplateBuilder = restTemplateBuilder;
        //this.specificProductUrl = fakestoreurl;
    }
}

```

App ppty: use it so you ave common configuration. Generally DB url, ppties we will see in JPA.

When you run an enterprise app. Restart is not a good approach. When we change the ppty file. In that case we usually have a configuration service. It host all conf like service. We get the conf via 3rd party. We keep refreshing at certain duration. If you change the conf value in conf service. Then my service don't need to restart. After 15 min my service get the latest configuration... when we configure the values, we can specify the env wise...

All depend on key-value.

Spring cloud, actuator etc we will need? Read about them as well. are we going to study config server?

Protobuf:

Agenda

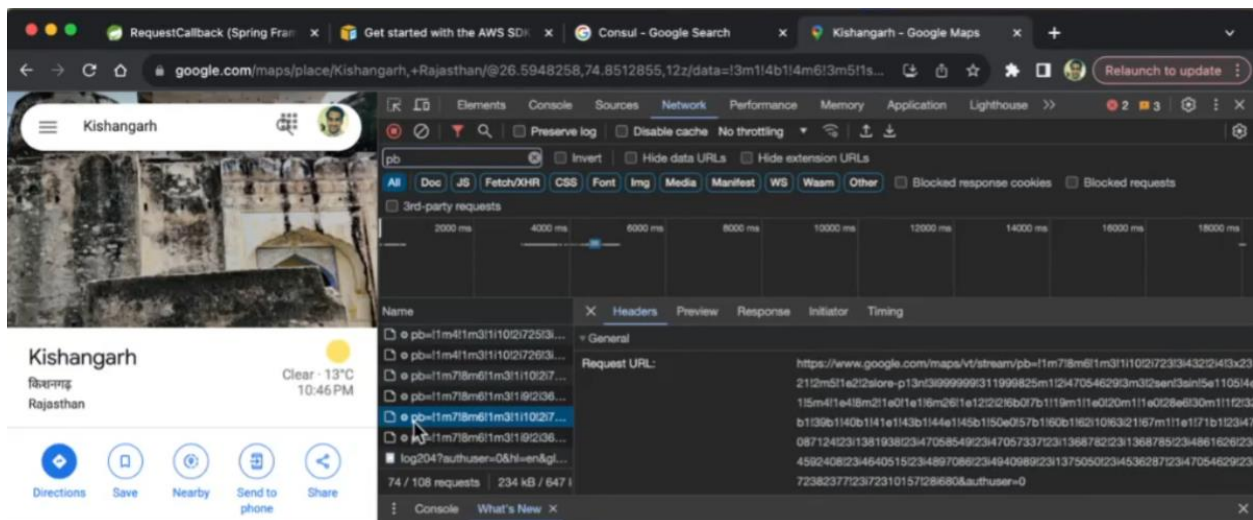
- 1) Delete Api
- 2) Implement Client
- 3) JSON | Protobuf
- 4) Application properties

Will start
9.10 pm

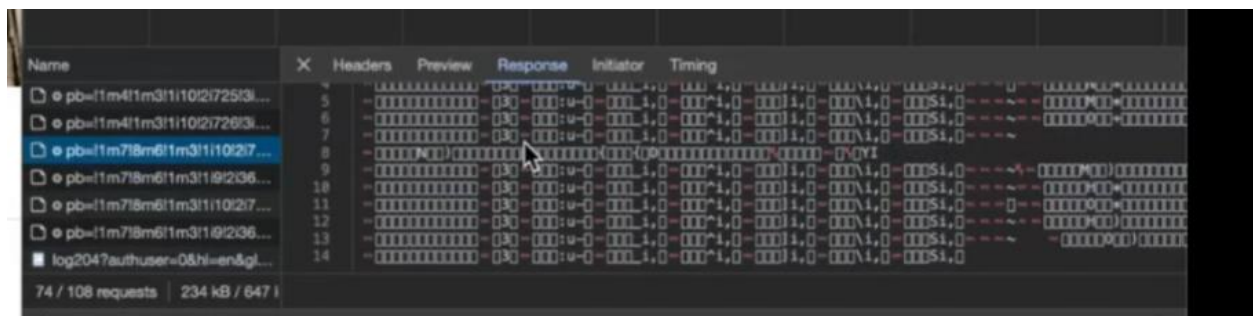
The screenshot shows a REST client interface with a GET request to `http://localhost:8080/products/3` sent successfully. The response is in JSON format, showing details for a product with ID 3, titled "Mens Cotton Jacket".

```
{
  "id": 3,
  "title": "Mens Cotton Jacket",
  "description": "great outerwear jackets for Spring/Autumn/Winter, suitable for many occasions, such as working, hiking, camping, mountain/rock climbing, cycling, traveling or other outdoors. Good gift choice for you or your family member. A warm hearted love to Father, husband or son in this thanksgiving or Christmas Day.",
  "price": 55,
  "category": {
    "id": null,
    "name": "men's clothing"
  },
  "allowedUser": null
}
```

Data we are getting is in which format? In JSON format. We are sending -getting in JSON format.



We have lot of pb. If u click on response tab:



We cant read the responses.

Not able to make anything out this response..

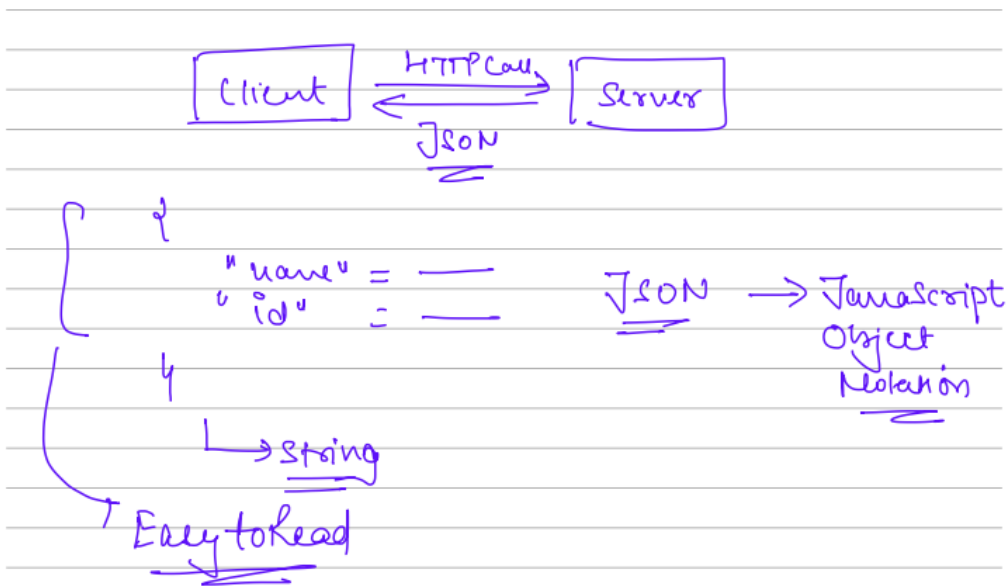
When you are amkinga call from one service to another service, there is a client..and a response back and a server.. here we are making http call and getting JSON response. Its nothing but a key-value pair..

JSON is like a string and easy to read...

When you are passing the data over a network.. what kind of data goes in? a binary data which goes over network..

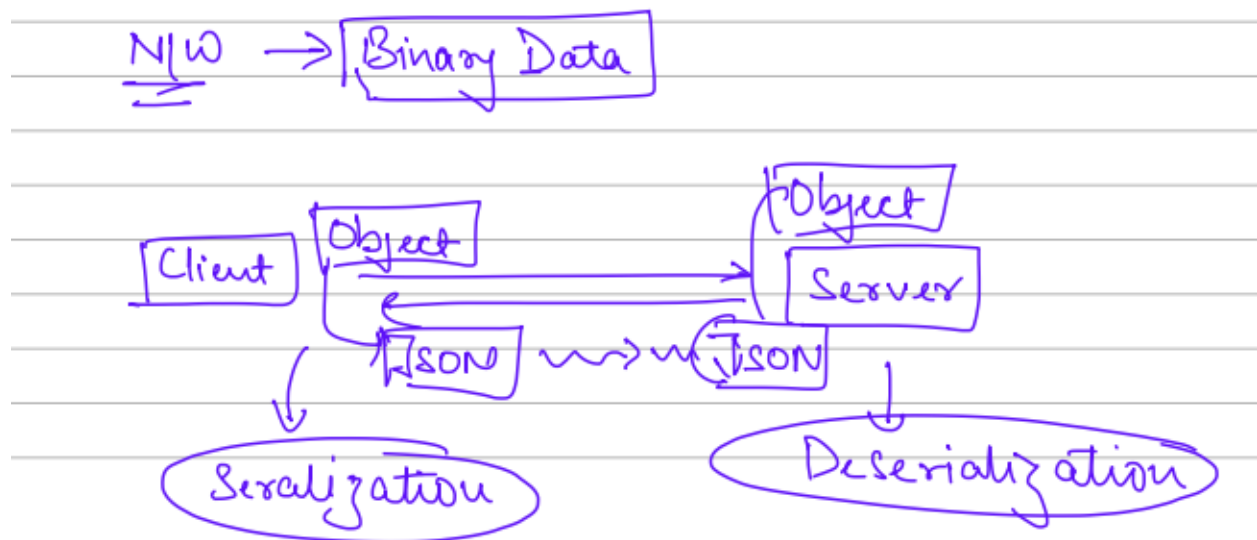
Then we try to convert to readable format...

When you are sending froma client.. it's a object..someone just do conversation on behalf of client. The object converts to JSON.



When you are sending from a client.. it's a object..someone just do conversation on behalf of client. The object converts to JSON. JSON will flow and server receive a JSON and that JSON will get converted to a object again at server end..

We are talking about binary data not bytestream



This process is called **Serialization** and JSON to object conversation is called **deserialization**..

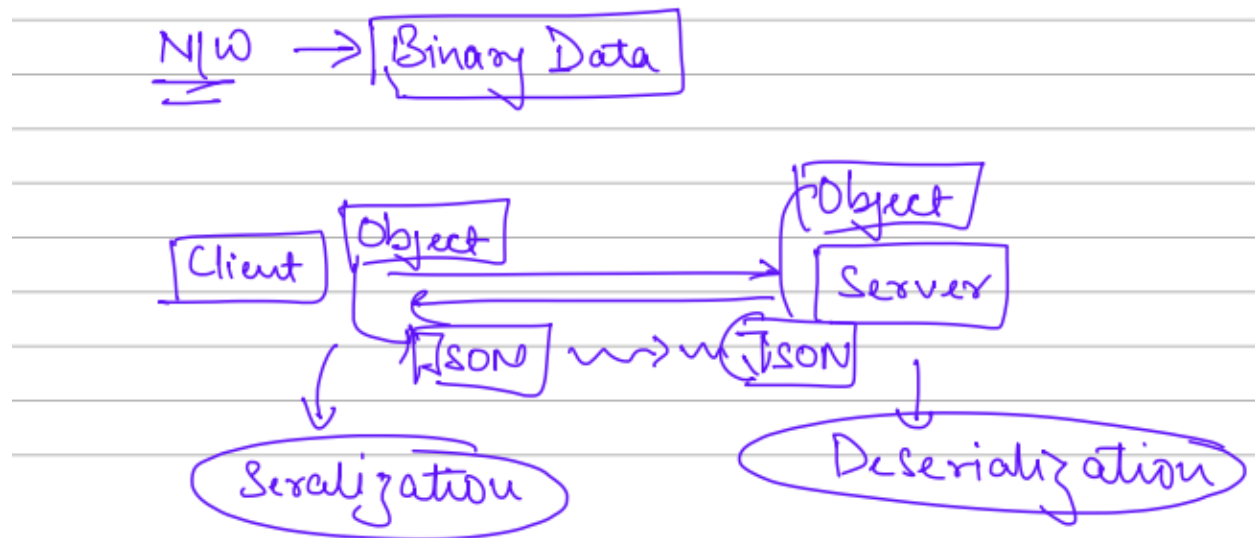
This process is expensive process as we are converting obj→json→obj.. **this expensive process hampers performance**... if I send the data directly in **binary format that will be more performance** oriented.

That's what protobuf does...

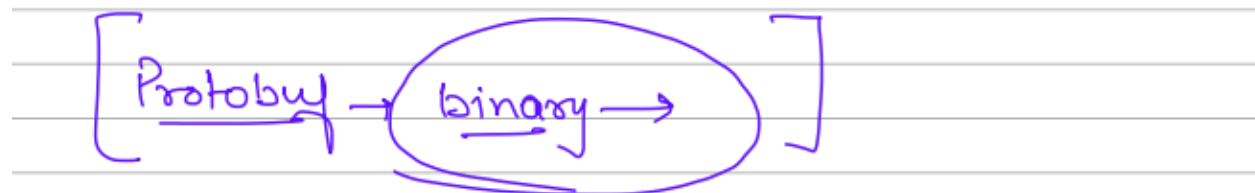
<https://protobuf.dev/>

Protobuf converts the data into binary and send it over network. it Reduces one layer of computation. Hence protobuf is faster than JSON.

can we say it compresses the info? Yes.. com and decom but it avoid the conversation..



Hamper performance



but how they convert pojo to binary? Look urself.

Protobuf provides direct conversation to binary and ahnce data travel faster. Less compute heavy. Also there is no serilization and deserilization.. protofile is a file where you will define the structure in protobuf format...

definetly we are gong to study grpc client because grpc uses it.....

Latency: protobuf.. majorly when you want lower latency.

most use case people don't use..99% me JSON is fine. 1% use case you use protobuf..

Security: by encrypting the data...

Q: Diff between client and template:

Client use template. Client is wrapper over rest template. When client does it. It fasading and give us simple method to work with. Clinent is a wraoper ovr template.

Does PB convert again?

First Object → JSON → binary.. over network bonary travel.. when binary convert to json. And json convertto obj.. if we can convert obj to binary one step is reduced.

Mohsin: kafkatemplate: still calling over network..? when you connecting that. Kafka nothing but a service. At the end you are making http call.. when you are running kafka... when you are connecting to Kafka template I don't bother what kind of .. kafka tem act like a client..

Kafka template is a template created on rest template... kafka template nothing but a kafka client... template they created over rest.

Same for JDBC template: anywhere when you don't have to deal with http client.. rest template and how http is making call... all are client.. you don't worry about how url is created...how aconnection is getting created..

We are just using API provided

1. Kafka Template and JDBC Template:

- In the context of Apache Kafka and JDBC (Java Database Connectivity), "templates" are provided as part of their respective libraries. These templates abstract away the low-level details of interacting with Kafka topics or databases.
- The Kafka Template provides a simplified interface for producing and consuming messages to and from Kafka topics.
- Similarly, the JDBC Template provides a simplified interface for executing SQL queries and interacting with databases.
-

2. **Kafka Template as a Client:**

- Mohsin likens the Kafka Template to a client in the context of network communication. Just like a REST client (e.g., RestTemplate) makes HTTP calls to interact with RESTful services, the Kafka Template acts as a client for interacting with Kafka clusters...
- When developers use the Kafka Template, they don't need to worry about the underlying mechanisms of how Kafka connections are established or how messages are produced/consumed. They simply use the provided API to send/receive messages to/from Kafka topics.

3. **Abstraction over Network Communication:**

- The discussion emphasizes that whether it's Kafka, HTTP (using RestTemplate), or JDBC, the templates provided by these frameworks abstract away the details of network communication.
- Developers interact with these templates as clients, without needing to understand the intricacies of how connections are made, URLs are constructed, or how data is serialized/deserialized.

4. **Focus on Using the Provided API:**

- The main takeaway is that developers should focus on using the API provided by these templates (e.g., Kafka Template, JDBC Template) to perform the desired actions (e.g., produce/consume messages, execute SQL queries).
- They don't need to delve into the internal workings of network communication or protocol specifics; instead, they rely on the abstraction provided by the template to interact with the underlying system.

In summary, the discussion highlights the role of templates in abstracting away the complexities of network communication and providing a simplified API for developers to interact with underlying systems such as Kafka and databases. Developers can treat these templates as clients, focusing on using the provided API without needing to understand the implementation details of network communication.

Data At move is encrypted by https ? in both REST JSON and Protobuf//// when we make call on postman.. we make call on http.. doesn't matter .. independent of what kind of call you make..

Affan: obj converted to json.. this conversation..over network data goes as bin regardless it has come from JSON or protobuf.. that process called serilization...???

Serilization: obj → JSON... not the json to bin..

JSON→ Obj is deserilization..

Json will be transferred in binary format...

Abhishek:

But if I convert object to binary that's faster... when you transfer obj to DB or anywhere it will be in binary format.. that object is directly transferred to bin. JSON is become to popular.. initil level was obj-binary...

It can be simple object... it will convert to bin and transferred?

Nikhil: object to JSON is expensive and same to obj—XML.. is expensive.

JSON is nothing but a string object... then it gets convert to binary...

WHay protobuf:

A client is n java app.. another on C++

If I converted java directly to binary I cant read to that in C++.. not easy.... That's why JSON serilization and deserilization comes..

To make it lang agonstic we need standard format of conversation.... JSON...

All prime we have in java not in C++. Can be issue with conversation...

Protobuf: I am not worried about java or C++. You give me proto obj I will use to convert to binary and convert to proto obj...

But java object also needs to be converted to protobuf

but object can be cobnvert to xml also

Mohsin: If I implement protobuf in c++. Same for java. When they interact they will botehr about whether I am getting protobuf or not...

Protobuf is independent of language... obj→ protobuf and makng it travelll..

Prtotobuff generally with on GRPC call..

