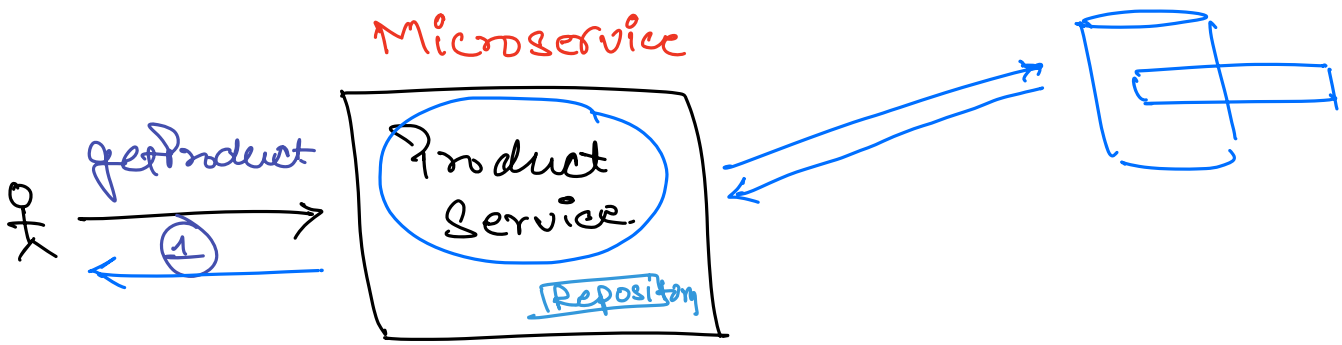
Agenda

1) Intro to fakestore.
2) Build our first Proxy APIs.

=> Productservice.

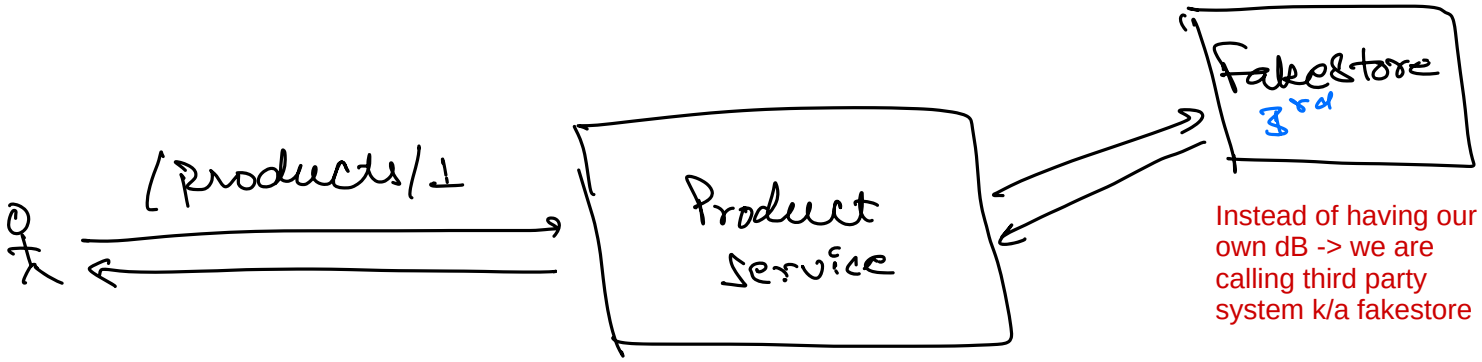
- create a Product
- get a Product
- update product
- delete

} CRUD



Demo API to call 3rd party.

Here we are not implementing our own db



Instead of having our own dB -> we are calling third party system k/a fakestore

⇒ fakestore : 3rd party system that provide sample API's related to Ecommerce.

⇒ Implement a product service which uses a 3rd party API behind the scenes to get the work done.

Our aim is to learn how to call 3rd party API and not implementing our own db

⇒ Learn how to call 3rd party APIs from our codebase.

⇒ Sample Product API's to get data from 3rd party system.

⇒ LLD of our Service - for calling 3rd party APIs what all requirement we want

1) Requirement Gathering

⇒ CRUD APIs.

2) Class Diagram.

↳ Product.

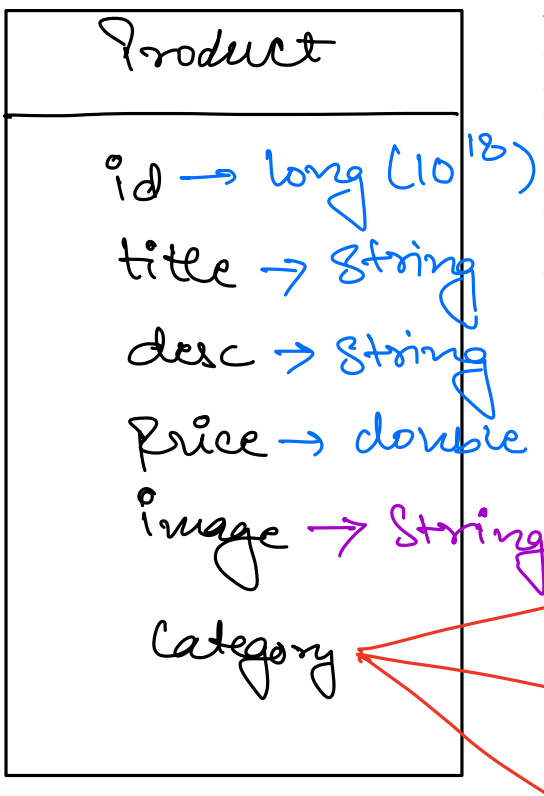
enum: is not good way to define datatype for Category. Since enum has fixed #of constants and category is something which keeps on adding depending upon the vendor. The vendor can place his product in new category as well. So changing enum requires code changes and frequent code changes is not recommended just for adding a category. Try to avoid code changes in production.

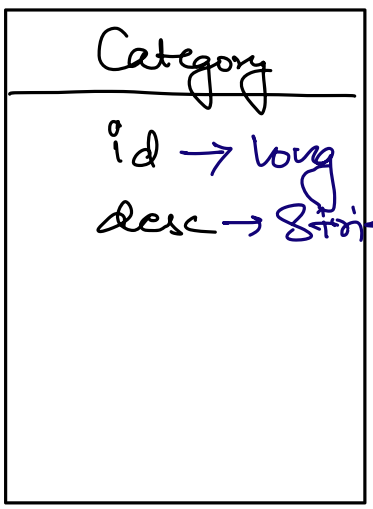
Plus you might have 10,000 category down the line, generally these big enums are not acceptable

String is also not a good way: Lets say there are a lot of products say a shirt so for lot of products, category shirt will be same and due to redundancy it will consume a lot of memory.

Also if you want to compare two strings, you have compare character by character so if a string is of length n, comparison of string is slower

So we will provide admin the access to add additional Category, rather than a code change?





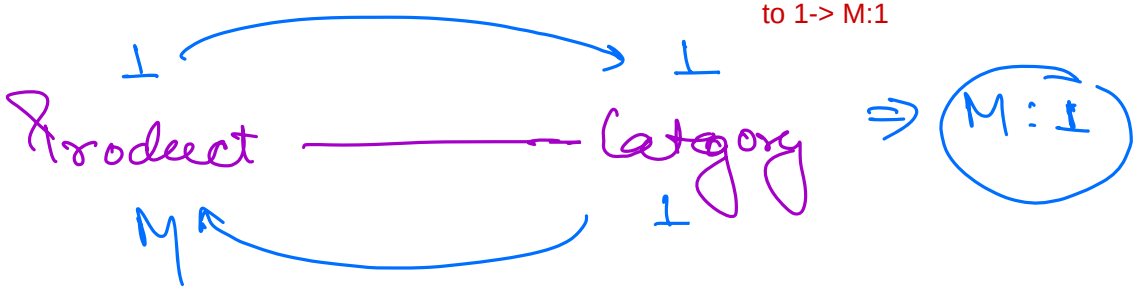
Enum ✗

String ✗

Category

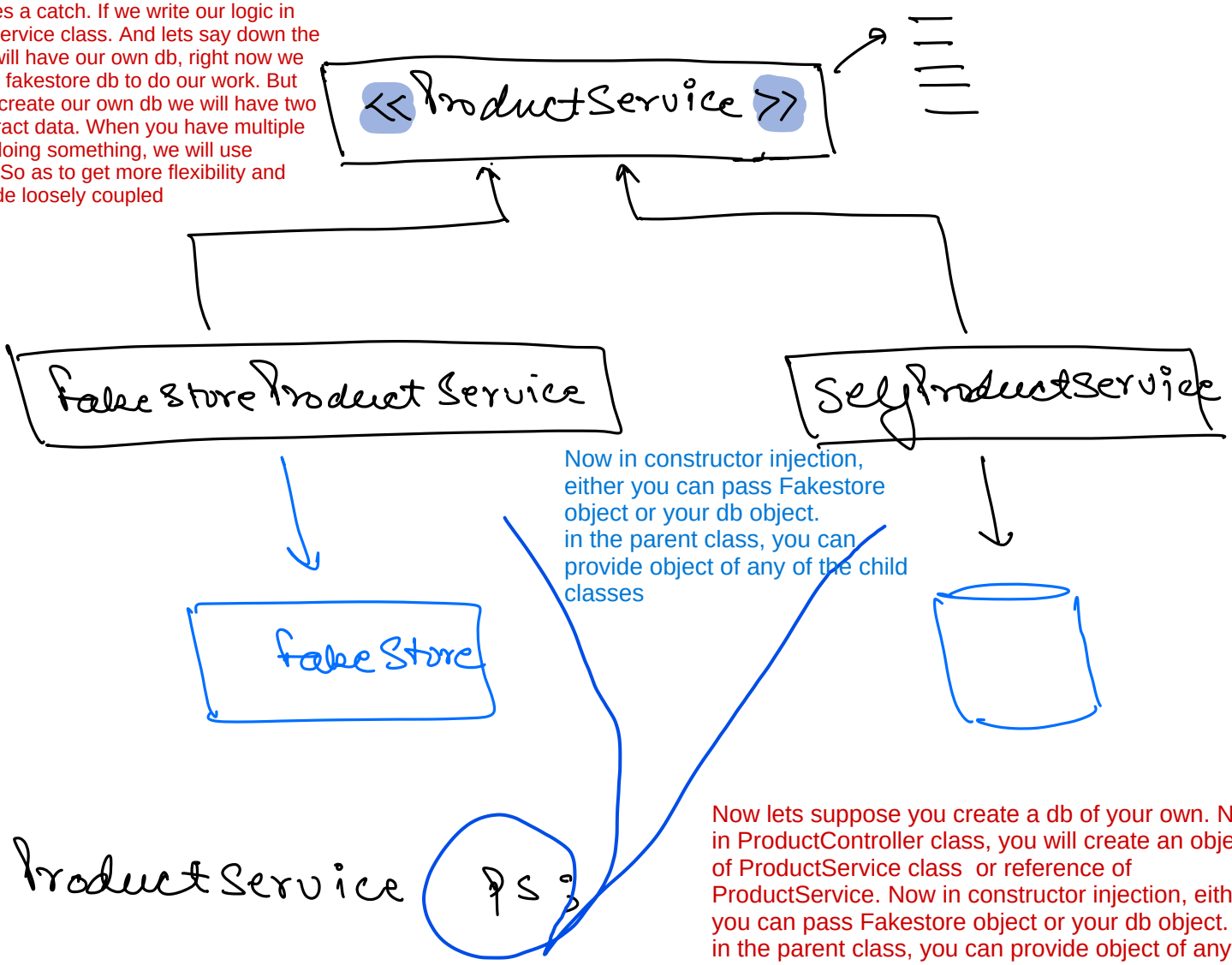
So we will take Category as a separate class and assign category as a foreign key so there will be no duplication

Cardinality will be Many to 1-> M:1



We have made ProductController class. Now to write business logic we will create ProductService class. Because here Controller is a chef which will give the request to chef

Now heres a catch. If we write our logic in ProductService class. And lets say down the line, we will have our own db, right now we are using fakestore db to do our work. But once we create our own db we will have two db to extract data. When you have multiple ways of doing something, we will use interace, So as to get more flexibility and make code loosely coupled



Now lets suppose you create a db of your own. Now in ProductController class, you will create an object of ProductService class or reference of ProductService. Now in constructor injection, either you can pass Fakestore object or your db object. in the parent class, you can provide object of any of the child classes

⇒ Program to Interface, not to implementation

⇒ loosely coupled.

Dependency Inversion principle --> two concrete classes should not be depending on each other.They shoudl depend via interface