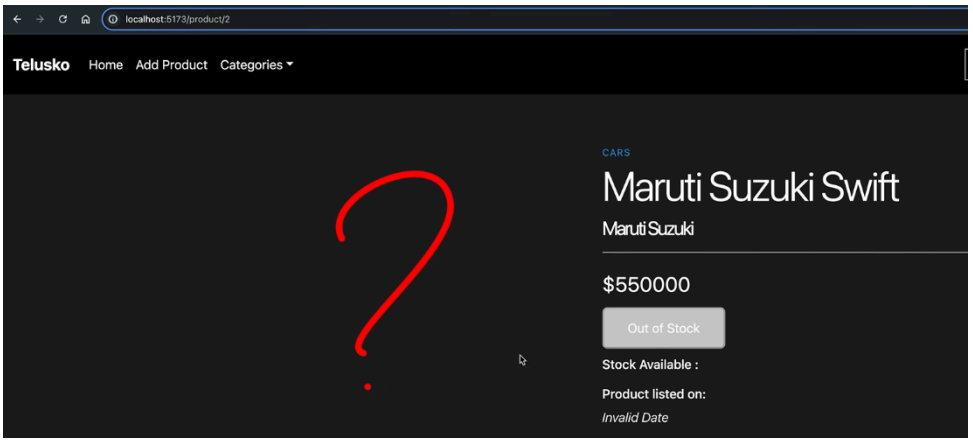
25-Add Product with Image

Now, we move to a more difficult and crucial part of our project: adding a product along with its image. In the UI frontend, we are getting all product lists, and upon clicking on each particular product, we expect an image beside the product. However, the implementation for adding a product with an image is pending on the backend side.



Adding Image Support in the Backend

1. **Updating the Backend Model:**
   - Currently, there is no variable assigned for an image in the backend model. We will add a variable for the image.
   - As image type is stored as large file data, we will annotate it with `@Lob.`
   - The @Lob annotation in Java is used to indicate that a particular field in an

```
37      private BigDecimal price;
38      private String imageName;
39      private String imageType;
40      @Lob
41      private byte[] imageData;
42  }
43
```

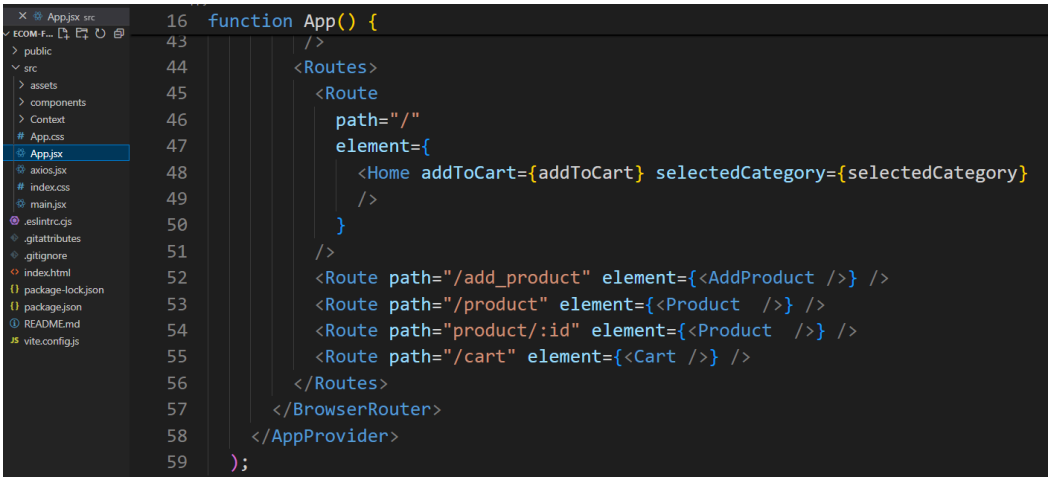
     entity should be treated as a "Large Object" (LOB) in the database. This annotation can be applied to fields that hold large amounts of data, such as text (CLOB) or binary data (BLOB).
   - **BLOB (Binary Large Object):** When the **@Lob** annotation is applied to a **byte[]** or a **serializable field**, it tells the persistence provider to store the data as a BLOB. This is commonly used for storing binary data like **images, videos,** or other **multimedia content.**

2. **Handling Date Format in UI:**

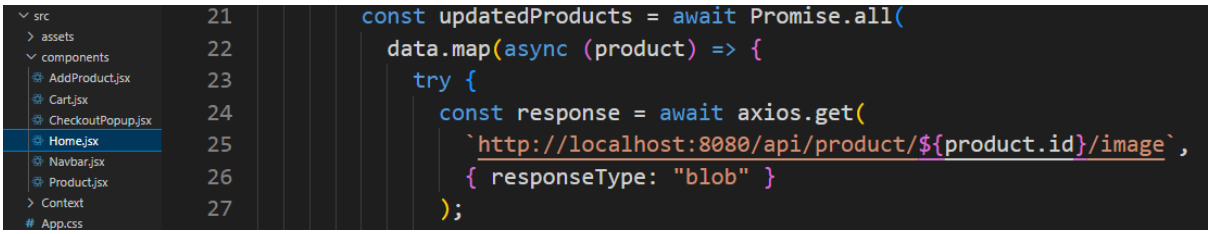
- Earlier, we discussed handling the date either on the backend or frontend. With the new updated UI, the date format has been handled on the frontend side.
- So, no need of changing the date format on the backend side.

3. **Adding Product Functionality in UI:**
   - The updated UI now places the image beside the product upon clicking a particular product.
   - The home page also includes an "Add Product" functionality, allowing the user to add product details, including the product image.

4. **Changes in `App.jsx`:**
   - The `App.jsx` file includes new components like `cartFeature` and `AddProduct`.
   - For now, we will focus on building our `AddProduct` feature.



5. **Fetching product and separately:**
   - The `Product.jsx` file has methods for fetching the product and image separately.



- Both `Home.jsx` and `Product` URL contain images and products along with id: `localhost:8080/api/product/{productid}/image`.

- **Backend URLs:**We will work with two URLs on the backend: one for the product and another for the image.

```
useEffect(() => {
  const fetchProduct = async () => {
    try {
      const response = await axios.get(
        `http://localhost:8080/api/product/${id}`
      );
      setProduct(response.data);
      if (response.data.imageName) {
        fetchImage();
      }
    } catch (error) {
      console.error("Error fetching product:", error);
    }
  };

  const fetchImage = async () => {
    const response = await axios.get(
      `http://localhost:8080/api/product/${id}/image`,
      { responseType: "blob" }
    );
    setImageUrl(URL.createObjectURL(response.data));
  };
```

Adding Image Data

1. **Model class variables:**
   - Add variables for image data in the model class.

```
18
19 @Entity
20 @Data
21 @AllArgsConstructor
22 @CrossOrigin
23 public class Product {
24
25     @Id
26     @GeneratedValue(strategy = GenerationType.IDENTITY)
27     private int id;
28     private String name;
29     private String descp;
30     private String brand;
31     private String category;
32     private boolean available_status;
33     private int quantity;
34     private BigDecimal price;
35     private String imageName;
36     private String imageType;
37
38     @Lob
39     private byte[] imageData;
40 }
```

2. **Updating the Controller:**
    - Create the `addProduct()` method in the `ProductController` class, which returns `ResponseEntity<?>` (wildcard) since we don't know what kind of data is being received from the client. It might be a product or a status code.
    - Use `@PostMapping("/product")` as the mapping request.
    - Use `@RequestPart` for handling multipart requests. This annotation will accept the request in parts (maybe 2, 3, or multiple) and for the image as `MultipartFile`.

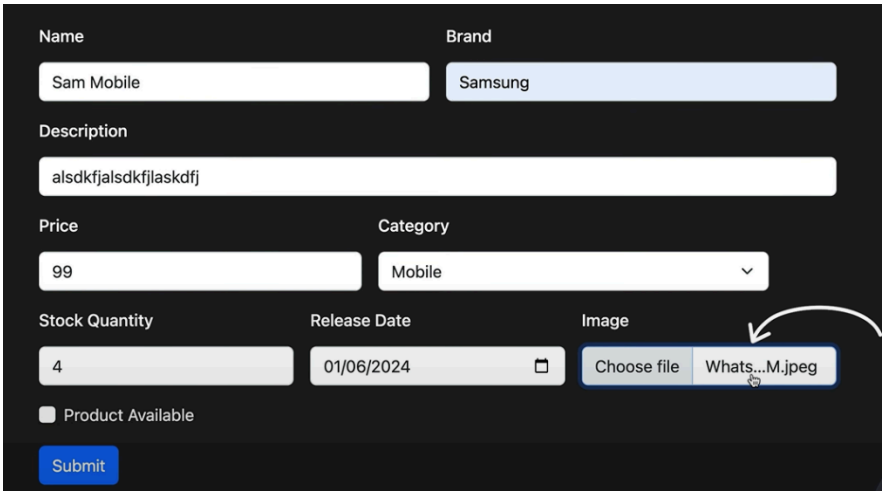
```
@PostMapping("/product")
public ResponseEntity<?> addProduct(@RequestPart Product product,
                                    @RequestPart MultipartFile imageFile){
    try {
        Product product1 = service.addProduct(product, imageFile);
        return new ResponseEntity<>(product1, HttpStatus.CREATED);
    }
    catch(Exception e){
        return new ResponseEntity<>(e.getMessage(),HttpStatus.INTERNAL_SERVER_ERROR);
    }
```

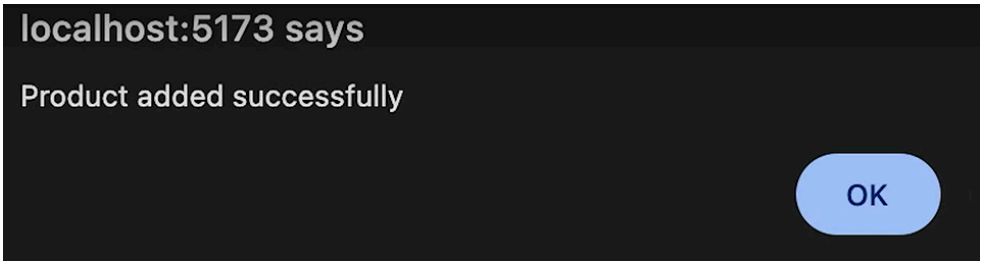
Verifying and Testing

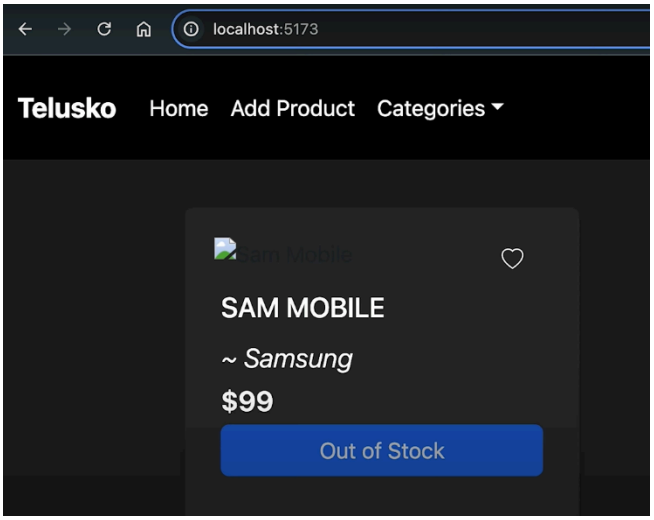
- **Service Layer:**
    - The `ProductService` class will use the repository to add the product through the `save` method from the JpaRepository. Here, we are sending the image along with the product.
    - Convert the image to byte format before storing it in the database.

```
public Product addProduct(Product product, MultipartFile imageFile) throws IOExceptic
    product.setImageName(imageFile.getOriginalFilename());
    product.setImageType(imageFile.getContentType());
    product.setImageDate(imageFile.getBytes());
    return repo.save(product);
}
```

- **Restarting the Project:**
    - Relaunch the project.
    - In the React app, click on the "Add Product" option.
    - Fill in the product details and submit the form. A popup window should display the message "Product added successfully."







- **Verifying in the database:**
  - Check the `h2-console` to ensure the image is stored in the database.