Uploading/Retrieving/Displaying Images

Front end

Getting a file selection

The input type **`file`** will show a file selection dialog which can be used to select a file.

```
<input type="file" id="file" ref="fileInput"/>
```

When a file is selected, the input element will receive a change event.

Vue provides a special way to access elements on a page. You can use the **`ref`** attribute to give the element a reference id which can be used to examine the element in your Vue code.

You can find the value of an element using the **`$refs`** property of a component.

You can find the file information for the element above using this syntax:

```
this.$refs.fileInput.files[0];
```

This value will be updated as the file selection changes.

POSTing the file data using axios

You can post the file info to the backend API using a **`content-type`** of **`multipart/form-data`**

To do this using axios, we can use the **`FormData`** object available in Vue.

Once you create a **`FormData`** object, you should append the **`'file'`** property to it with the value returned by the file select dialog as the value. Here's an example:

```
let formData = new FormData();
formData.append('file', this.file);
```

You can then post the FormData object to the API but you need to specify the correct **`content-type`**.

Axios allows us to add an object that specifies some extra option properties. One of the possible options is the **`headers`** option.

The **`headers`** option is specified using the **`headers`** property, which will contain an object with properties for each of the header values you want to add.

Here's an example of an options object with the needed header for our post:

```
const options = {
      headers: {
            Content-Type': 'multipart/form-data'
      }
}
```

You can then add the **`options`** object to your you axios post request:

```
axios.post('/upload', formData, options).then(....)
```

Now let's look at the back end code.

Back end

API Controller

The multipart form data properties are sent over as query parameters. You can capture these using the Spring **`@RequestParam`** annotation with a MultipartFile type variable. Here's an example for the post above:

```
@RequestMapping(path = "/upload", method = RequestMethod.POST)
public ResponseEntity<String>
      uploadFile(@RequestParam("file")MultipartFile file) { … }
```

Inserting binary data

PostgreSQL can store binary data such an image in a field of type **`bytea`**

For our purposes it will also be important to store the filename.

You can use the **`getBytes()`** method of the **`MultipartFile`** to get data which can be used for **`bytea`** data.

Here's an example:

```
public Integer save(MultipartFile file) throws IOException {
        String sql = "INSERT INTO image_data (image_name, image_data)
           VALUES (?, ?) RETURNING image_data_id";
        return jdbcTemplate.queryForObject(sql, Integer.class,
           file.getOriginalFilename(),
           file.getBytes());
}
```

Reading binary data from the database

Reading binary data using the **`jdbcTemplate`** is a little more complicated than reading other types of data we have used. The problem is that the **`jdbcTemplate`** doesn't have a method like **`getString`**, **`getInt`**, etc. for reading binary data. The built-in Java class **`ResultSet`** DOES have a method to do this. We can use the **`ResultSet`** method **`getBytes()`** to get back a byte array containing the binary data. In order to do this though, we need to allow the **`jdbcTemplate`** to use a **`ResultSet`** (rather than a **`SQLRowSet`**, etc.) to read the data. We can do this using the **`RowMapper<T>`** interface.

The **`RowMapper<T>`** interface requires implementing a method with this signature:

```
T mapRow(ResultSet rs, int rowNum)throws SQLException
```

We will look at how to use the **`RowMapper`** with the **`jdbcTemplate`** shortly but you can create a class which implements the **`RowMapper<T>`** interface and use the **`ResultSet`** it provides to read the binary data. Here's an example of what a method to read the binary data would look like:

```
@Override
public String mapRow(ResultSet resultSet, int i) throws SQLException {
    byte[] imageData = resultSet.getBytes("img_data");

    // convert to String
    String imageDataString = Base64.getEncoder().encodeToString(imageData);

    return imageDataString;
}
```

Once we have created a class that implements the **`RowMapper<T>`** interface, we can use the **`jdbcTemplate`** overloaded **`query`** method with this signature:

```
<T> T query(String sql, RowMapper<T> mapper, Object... args)
```

Here's an example of what this would look like if our **`RowMapper`** class is called **`ImageStringMapper`**:

```
public String getImageDataStringById(long id) {
    String sql = "SELECT * FROM image_upload WHERE img_id = ?";

    try {
```

```
        return jdbcTemplate.queryForObject(sql, new ImageStringMapper(), id);
    } catch (EmptyResultDataAccessException ex) {
        return null;
    }
}
```

The last pieces of what the back end needs to return is based on the image type.

In order to display the image correctly, the **`src`** attribute of the image needs some additional info.

The value that is dynamically set in the **`src`** attribute of the image tag on the page needs to have a prefix that describes the data - both the type of image and the fact that the data is base64-encoded.

For instance, for a jpg image, the prefix would be

`data:image/jpg;base64` followed by a comma and a space

For a png it would be

`data:image/png;base64` followed by a comma and a space

Given that, the result from the controller would be something like:

`" " + imageDataString`

Using the result String on the front end

Not much to do here… you can use the String built above in an img src attribute that is bound to the String returned from the above process.

You can now upload images, store them in the database, retrieve them, and display them!!!