Creating Custom Error Response and Exception

Why Use Custom Exceptions?

Although Java exceptions, as they are, cover nearly all exceptional cases and conditions, your application might throw a specific custom exception, unique to your code and logic.

Sometimes, we need to create our own for representing business logic exceptions, i.e. exceptions that are specific to our business logic or workflow. For example EmailNotUniqueException, InvalidUserStateException, etc.

They help application clients to better understand what went wrong. They are particularly useful for doing exception handling for different business logic constraints require different response codes to be sent back to the client.

Custom Checked Exception

Let's consider a scenario where we want to validate an email that is passed as an argument to a method.

We want to check whether it's valid or not. Now we could use Java's built-in IllegalArgumentException, which is fine if we are just checking for a single thing like whether it matches a predefined REGEX or not.

But suppose we also have a business condition to check that all emails in our system need to be unique. Now we have to perform a second check (DB/network call). We can, of course, use the same IllegalArgumentException, but it will be not clear what the exact cause is - whether the email failed REGEX validation or the email already exists in the database.

Let's create a custom exception to handle this situation. To create an exception, like any other exception, we have to extend the java.lang.Exception class:

```
public class EmailUniqueException extends Exception {
   public EmailNotUniqueException (String message) {
      super(message);
   }
}
```

By calling super(message), we initialize the exception's error message and the base class takes care of setting up the custom message, according to the message.

Now, let's use this custom exception in our code. Since we're defining a method that can throw an exception in the service layer, we'll mark it with the throws keyword.

If the input email already exists in our database, we throw our custom exception:

```
public class RegistrationService {
    List<String> registeredEmails = Arrays.asList("example@gmail.com", "one@gmail.com");

    public void validateEmail(String email) throws EmailNotUniqueException {
        if (registeredEmails.contains(email)) {
            throw new EmailNotUniqueException("Email Already Registered");
        }
    }
}
```

Now let's write a test client for our service. Since it's a checked exception, we have to adhere to the handle-or-declare rule. In the proceeding example, we decided to "handle" it:

```
public class RegistrationServiceClient {
    public static void main(String[] args) {
        RegistrationService service = new RegistrationService();
        try {
            service.validateEmail("abc@gmail.com");
        } catch (EmailNotUniqueException e) {
            // logging and handling the situation
        }
    }
}
```

Best Practices for Custom Exceptions

- Adhere to the general naming convention throughout the Java ecosystem - All custom exception class names should end with "Exception"
- Avoid making custom exceptions if standard exceptions from the JDK itself can serve the purpose. In most cases, there's no need to define custom exceptions.
- Prefer runtime exceptions over checked exceptions. Frameworks like Spring have wrapped all checked exception to runtime exceptions, hence not forcing the client to write boilerplate code that they don't want or need to.
- Provide many overloaded constructors based on how the custom exception would be thrown. If it's being used for rethrowing an existing exception then definitely provide a constructor that sets the cause.