TempBuddy Exercise Documentation

GitHub Repository: https://github.com/davidfrd/TBExcercise

David Fernando Redondo Durand
davidfrd2@gmail.com
09/09/2019

Endpoint (Billing GET)

Billing GET is the unique endpoint of the application. Its purpose is calculate the Billed Shifts detailing the rules applied for each shift and the result payrate.

Input

The body of this GET request contains a list of the shifts and other list with the rules that the program will applied.

In the following example, there is only one shift and only one rule that will be applied.

Example:

```
{
   "shifts":[
      {
         "id":1,
         "start":"2019-04-28 20:30:00",
         "end":"2019-04-29 00:30:00"
      }
   ],
   "rules":[
      {
         "id":1,
         "type":"FIXED",
         "start":"13:00",
         "end":"08:00",
         "payRate":10.50
      }
   ]
}
```

Output

After processing the request, and applying all rules to the shifts, the results must be the total pay and a list of the shifts with the rules applied to each.

- Pay: Total units to pay for all shifts
- BilledShifts
    - id: Shift ID
    - start: Shift start time
    - end: Shift end time
    - session: Shift duration

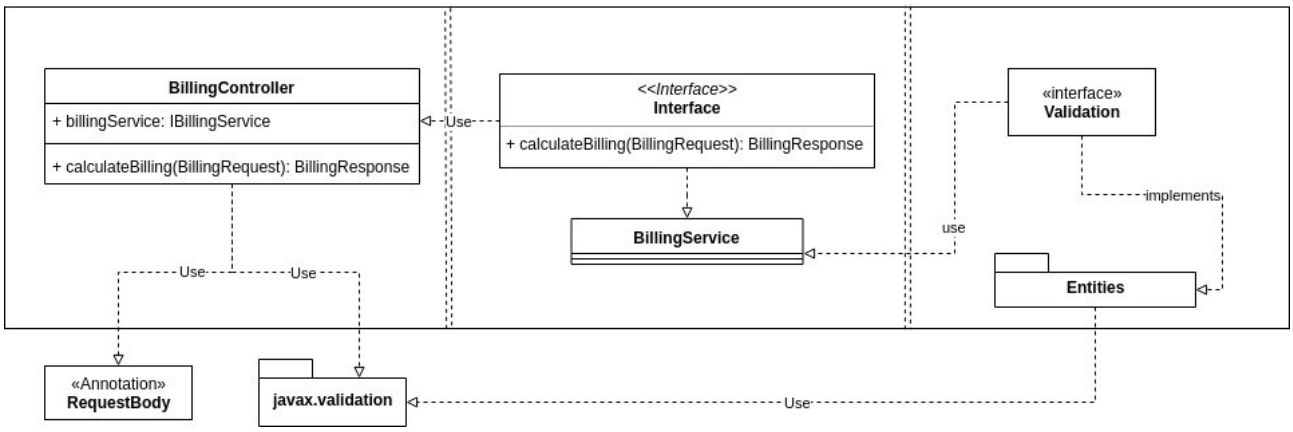
- pay: Total units to pay for this shift
- portions
  - id: Applied rule ID
  - start: Time when rule starts to be applied
  - end: Time when rule ends to be applied
  - session: Total time in seconds of the portion
  - pay: Total pay of the portion

Example:

```
{
    "pay": 42.00,
        {
            "id": 1,
            "start": "2019-04-28 20:30:00",
            "end": "2019-04-29 00:30:00",
            "session": 14400,
                {
                    "id": 1,
                    "start": "2019-04-28 20:30:00",
                    "end": "2019-04-29 00:30:00",
                    "session": 14400,
                    "pay": 42.00
                }
    "billedShifts": [
            ]
        }
    ]
}
            "pay": 42.00,
            "portions": [
```

Code Documentation

When you call the billing get request with the shifts and the rules in the body, the controller will catch the request and converts the *JSON* body to the *BillingRequest* using *@RequestBody* annotation while it validates all incoming data (without applying any business rule) using *@Valid* annotation. When this process is over and okay, the controller calls the injected *BillingService*, which will validate the request applying business logic and process the incoming data and finally, generating the *BillingResponse*.



Data conversion and non business logic validation

@RequestBody annotation transform the incoming JSON from request body to *BillingRequest* class with all shifts and rules within their respective classes. All this classes are validated with *javax.validation* library, which with the *@Valid* annotation allows us to apply and check all the rules defined in each class.

There is an example of the validation rules (*NotNull, NotEmpty and Positive*):

```java
public abstract class BillingRule implements Validable, Serializable {

    private static final long serialVersionUID = -4881567644190448278L;

    @NotNull
    @Positive
    protected Integer id;

    @NotNull
    @NotEmpty
    protected String type;

    @NotNull
    @Positive
    @JsonSerialize(using = DecimalWithTwoDigitPrecision.class)
    protected Double payRate;
```

Billing rule conversion

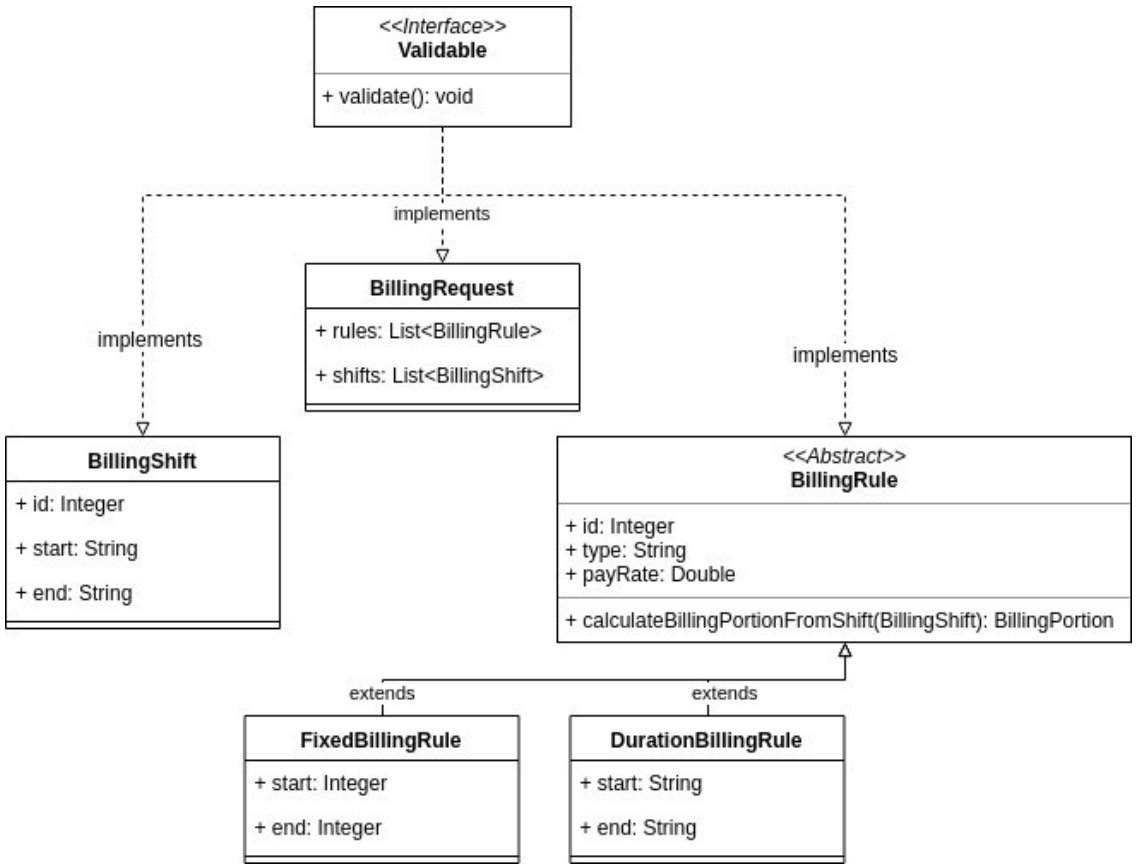
The billing rules have a more complicated conversion because it has two types (duration and fixed) with different type of data (start and end), and different business logic too.

To make the conversion, we need to apply *factory pattern* in this case, which, taking the type of the rule, decides what subclass should have this rule.

If the rule is *"FIXED"* type will create an *FixedBillingRule* object, and otherwise, if it's *"DURATION"*, it will create an *DurationBillingRule* object.

Code fragment:

```java
@JsonTypeInfo(use = JsonTypeInfo.Id.NAME, include = JsonTypeInfo.As.PROPERTY, property = "type", visible = true)
@JsonSubTypes({
    @JsonSubTypes.Type(value = FixedBillingRule.class, name = "FIXED"),
    @JsonSubTypes.Type(value = DurationBillingRule.class, name = "DURATION")
})
public abstract class BillingRule implements Validable, Serializable {
```



UML of the BillingRequest

Business logic

Validation

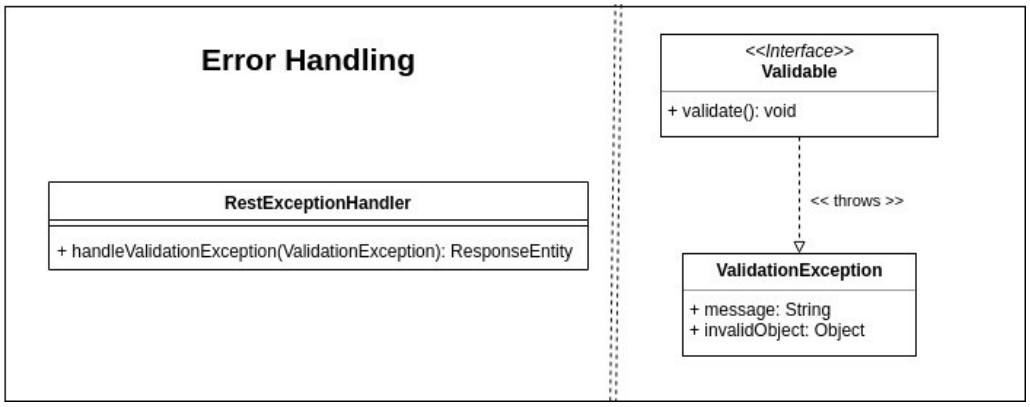
To implement the business logic validation, classes should implement the *Validable interface*, which provides an *validate()* method where you should put your validation logic. If the validation fails, you should throw a *ValidationException* with a message and the object that produced this error.

Applying rules

As we say in previously chapters, the rules are in two classes. Each class implements an method called *calculatedBillingPortionFromShift( )*, which takes an shift and return the portion calculated by applying the rule. But, there is a business logic shared between fixed an duration rule, and you can convert one to the other to implement this logic once. In this case, to simplify the business logic, the *FixedBillingRule* will be converted to an *DurationBillingRule* and process the shift as it were a duration rule.

Error Handling

The *ValidationException* will be caught by the *RestExceptionHandlerClass*, which is responsible for convert it into *CustomValidationErrorResponse* and return it to the client.



Error handling diagram