Software design specification document
Cairo University - SCS355

Team members
Ahmed Wael Nagy Wanas - 20206008 - ahmedwaelwanas@gmail.com
Adham Hazem Fahmy Shafei - 20206011 - adham852002@hotmail.com
Omar Adel Abdel Hamid Ahmed Brikaa - 20206043 - brikaaomar@gmail.com
Ali Esmat Ahmed Orfi - 20206123 - aliesmat612@gmail.com

Table of content

1. Introduction and system overview

Following are some aspects of the system that might need additional explanation in addition to the code and diagrams, the rest should be clear:

1.1. Authentication

Authentication on the system is done using JWT tokens. The response to a valid POST request to /login contains a "token". This token should be included in the auth header of the subsequent requests to the system; otherwise, the request will return 401 unauthorized status. If you are using our Postman collection, you do not need to worry about setting the header manually since we have a script that sets it after the /login request.

1.2. Authorization

Authorization is done using the isAdmin payload in the JWT token. If the token has the isAdmin claim set to false, the authorized requests will return 403 FORBIDDEN status. The system is initially populated with an admin account with the following info:

```
{
   username: admin,
   email: admin
   password: admin,
}
```

1.3. Interceptors

We make use of Spring MappedInterceptors to perform authentication and authorization checks before controller actions. These can be found in the payments.controllers.interceptors package.

1.4. Datastore

We made our own little framework that handles data storage in memory. It supports DBMS-like operations like select, update and insert and makes use of lambda expressions that were introduced in Java 8 to do these operations declaratively. We chose to take this route to avoid relying on third-party data storage software that might not be present on the TAs machines. However, using such third-party software and frameworks to handle data storage would be a better option. The datastore framework can be found in the datastore package.

1.5. Payments, services, service providers

We think of them this way:
- Services and service providers are dynamic entities that get added to the database
- Handlers are static assets in the code (typically these would be on the third-party processors' machines).
- Handlers are responsible for handling the payment request and sending a response containing the amount that should be deducted.
- How do dynamic entities communicate with static assets?
- Using an abstract class, we force these static assets (Handlers) to provide a handler name, handler request keys and handler constraints.
- The admin who adds the service provider specifies the name of the handler that the provider is going to use.

- The user paying for a service to a service provider must specify a "handlerRequest" attribute in the request body which is a key-value pair that includes all the handler request keys. The values of these keys must comply with the handler constraints.
- An example of a handler name is "WE_INTERNET"
- An example of a handler request key is "landline"
- An example of a handler constraint is "the landline number must be 8 digits"
- The memory is initially populated with services and service providers in the assignment pdf.
- The admin can add new services and service providers.

1.5. Entities

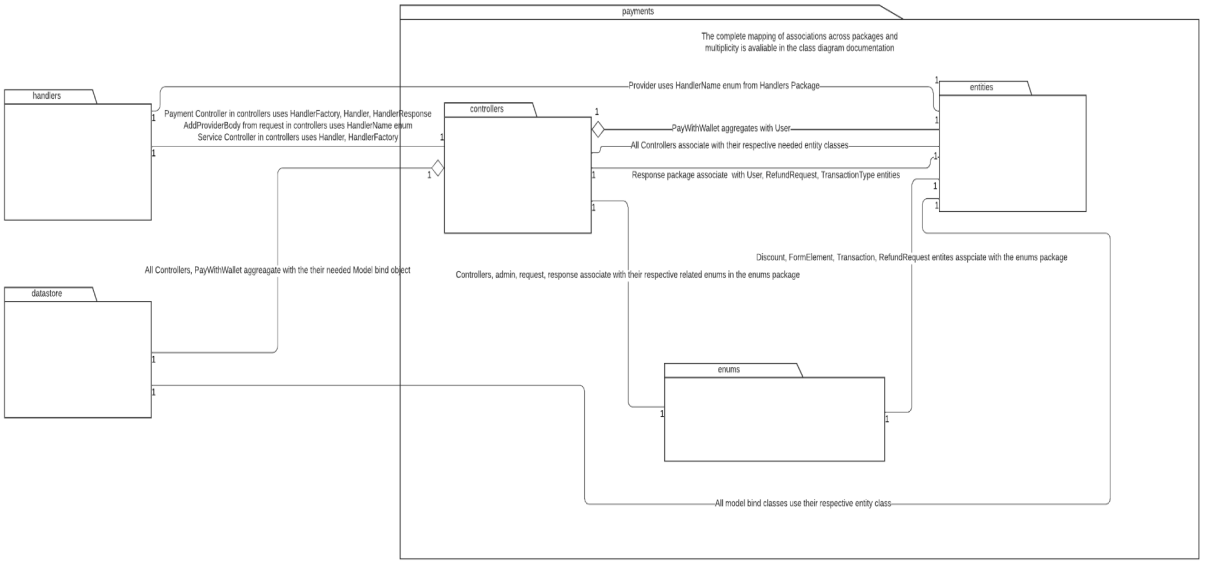
Entities are organized in a relational model. There is no aggregation between an entity and another. They only relate to each other using primary keys and foreign keys and hence this relationship will not be visible in the class diagram. It can be represented using an ERD diagram which is not required. The datastore Model class and the controllers are responsible for doing the joins between the entities.

2. Subsystem decomposition and class diagram

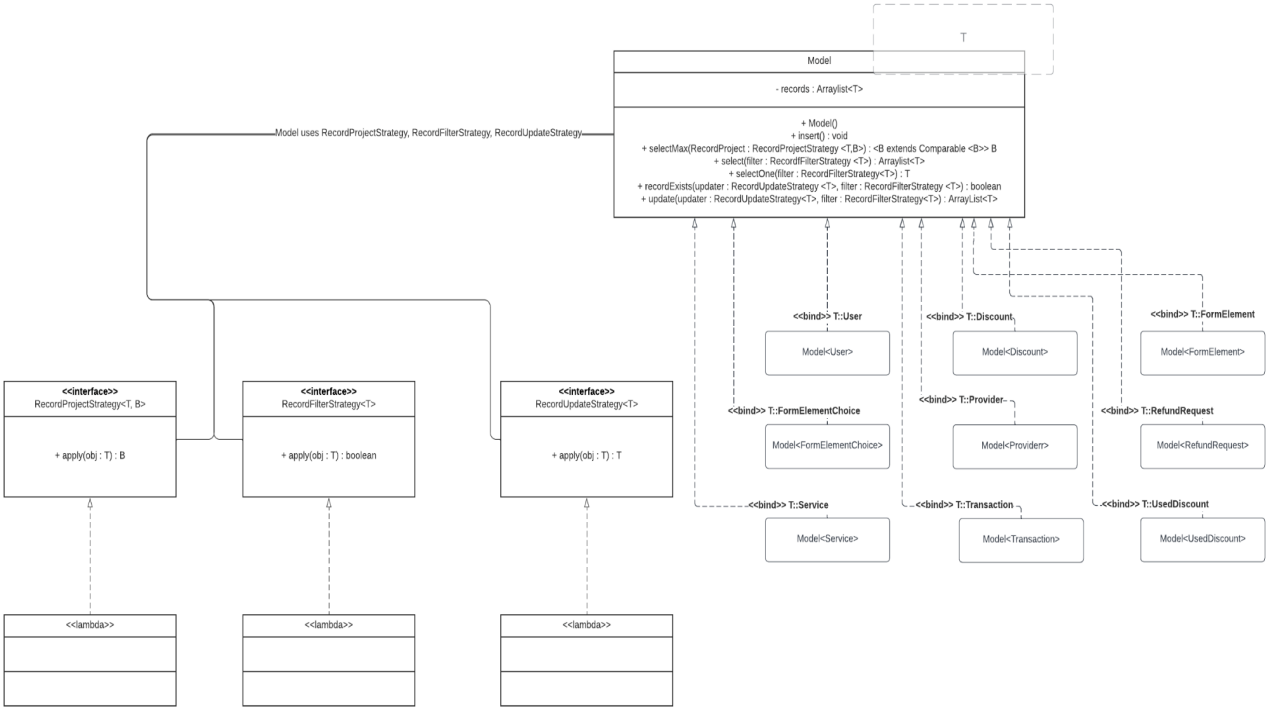
Note: Inside each package some classes might appear "floating" (without any associations or whatsoever). This is because they associate with classes from other packages. Including a picture of the class diagram as a whole would make it unreadable and hence we chose to divide it into packages and provide explanations of the associations of the classes in the following section.

2.1 High level packages

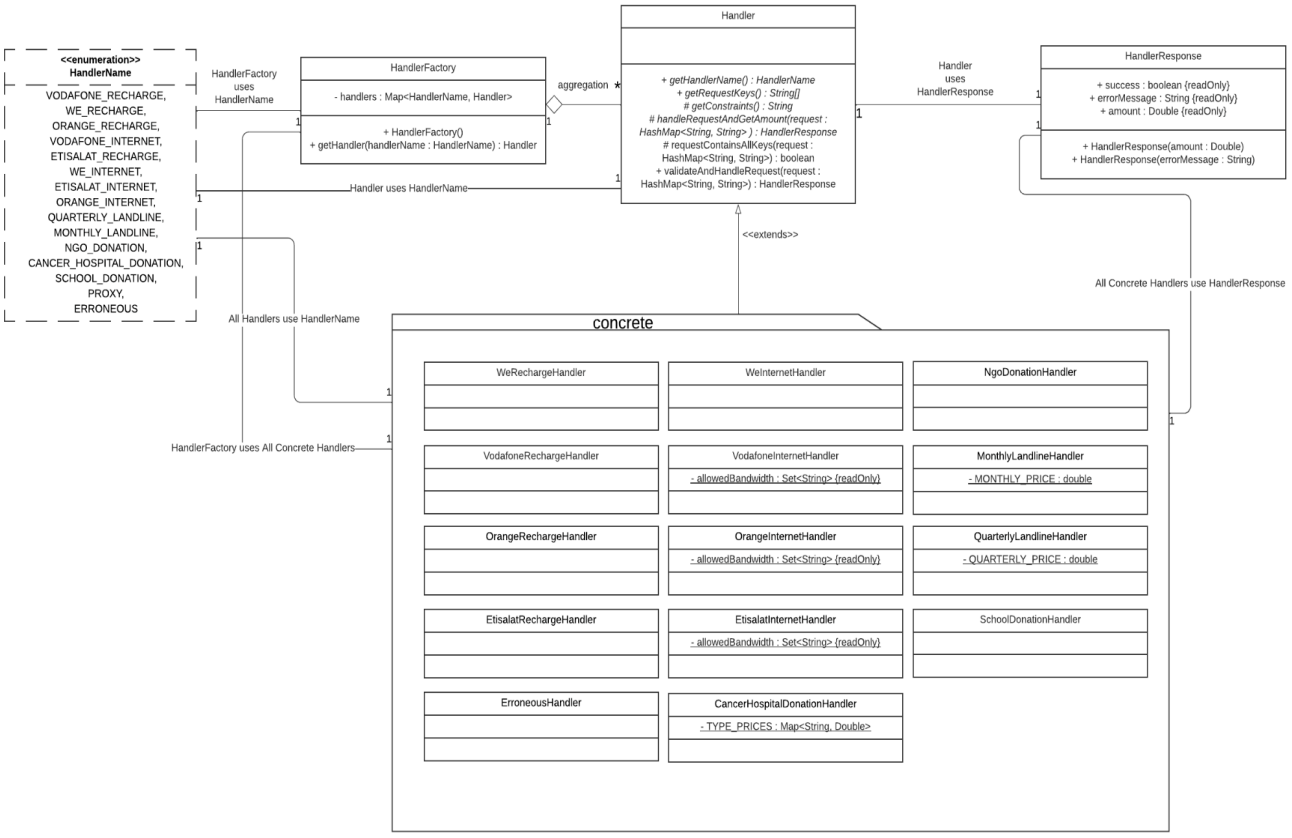
Only shows packages and sub packages and how they interact with each other



2.2 datastore package

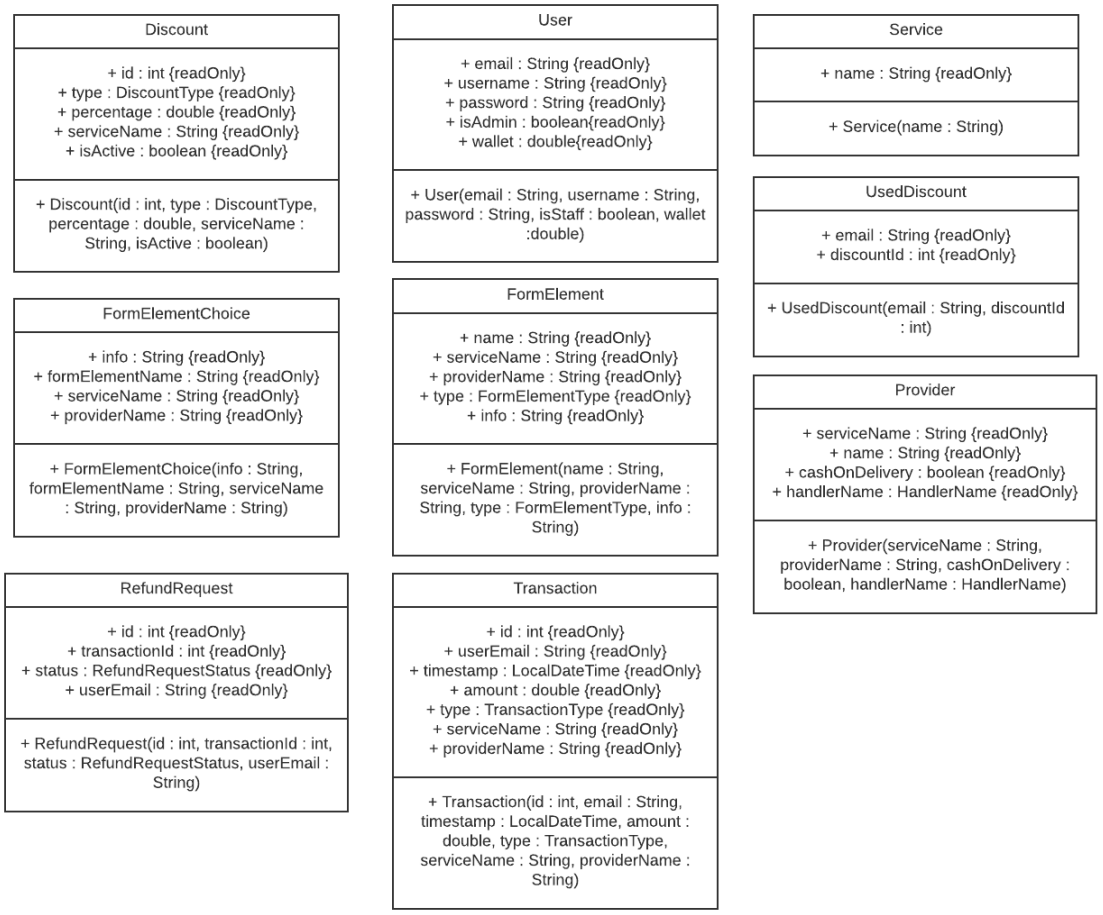


2.3 handlers package

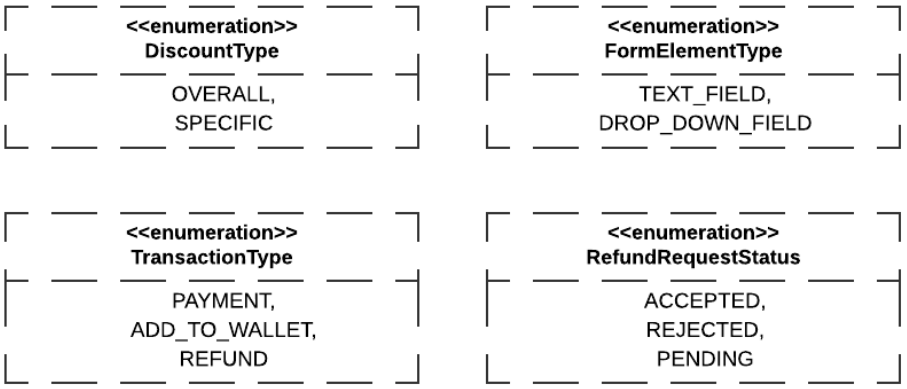


2.4 payments.entities package

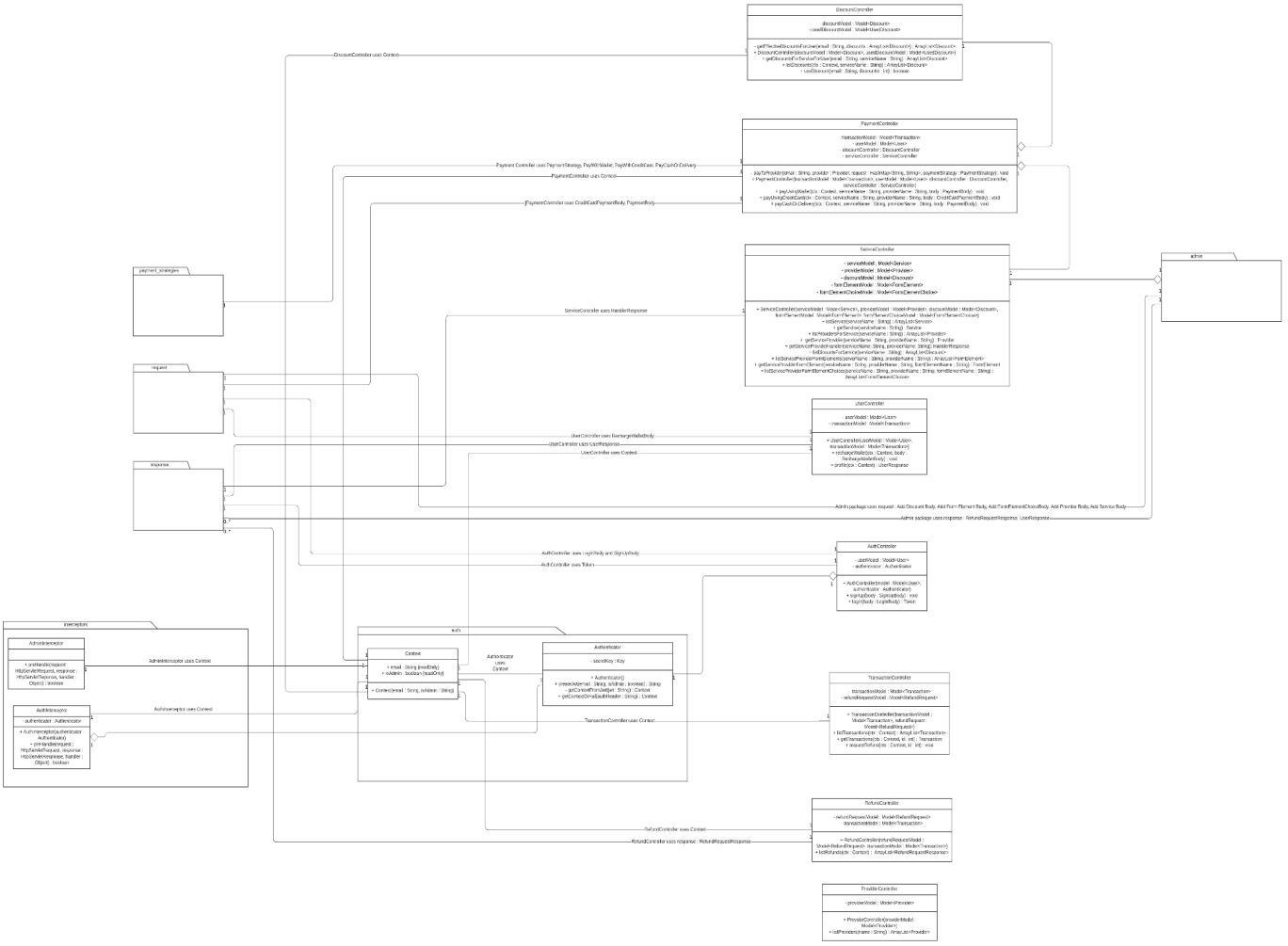
No entity aggregates with another since we are using a relational model where relationships are done using primary keys and foreign keys.



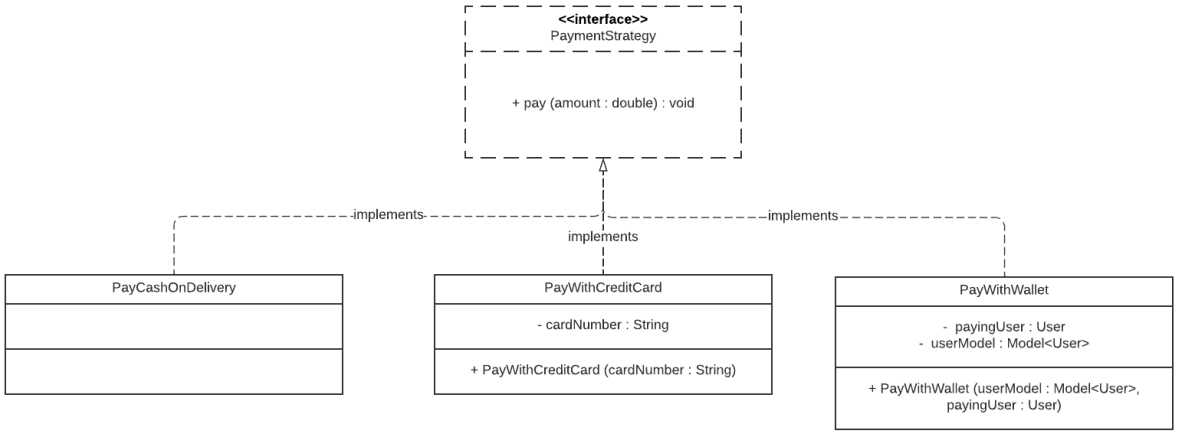
2.5 payments.enums package



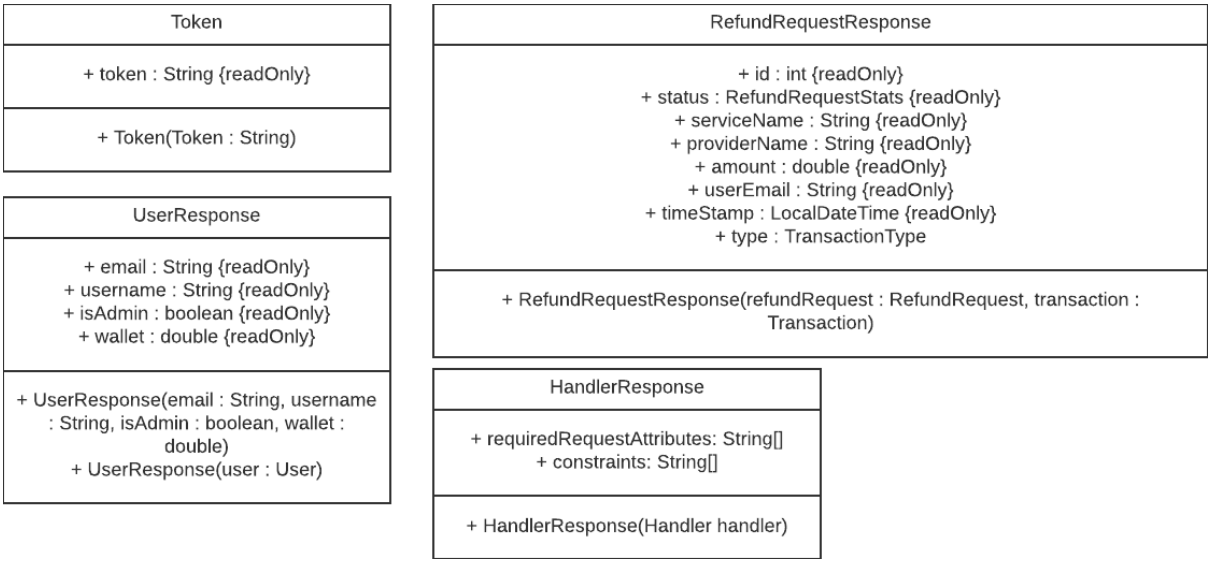
2.6 payments.controllers package



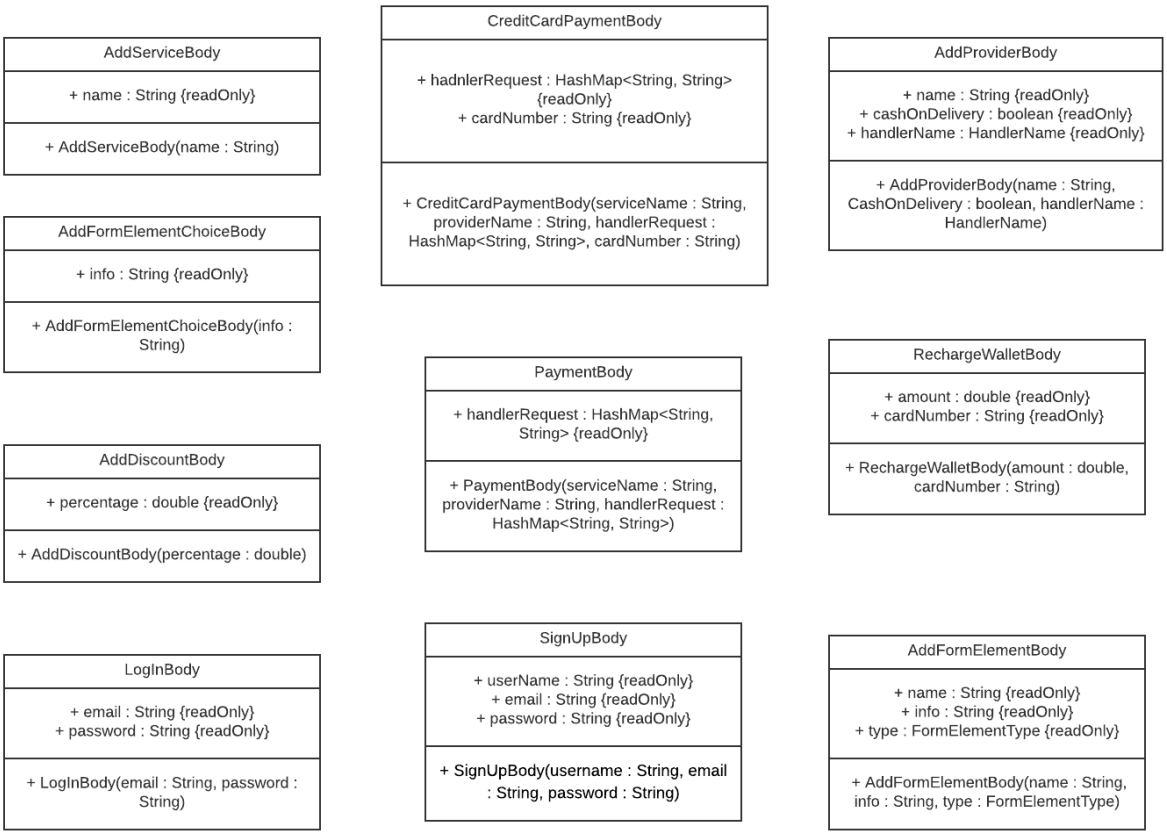
2.7 payments.controllers.payment_strategies package



2.8 payments.controllers.response

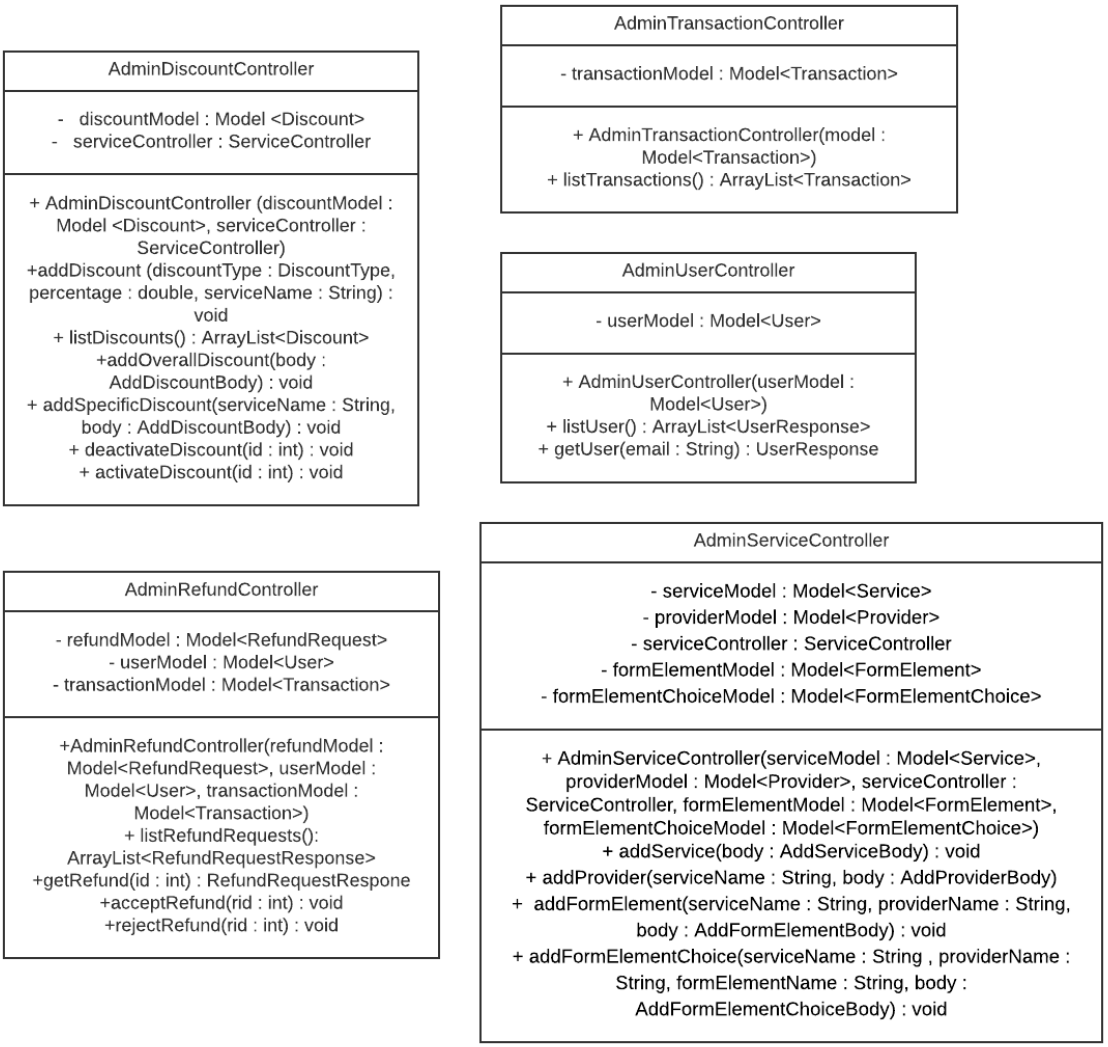


2.9 payments.controllers.request



- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- Any relations reaching into this package have been mapped in previous sections.

2.10 payments.controllers.admin package



- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- No relations reaching into this package.

Associations
The following are associations between classes. Some could not be represented in the diagram because they referenced classes from outer packages, and the diagram only zoomed into packages one by one.

2.1 & 2.2 High level packages and datastore package:

2.1.1.1) bind class “Model <User>”:
- Associates with User entity in a 1 to 1 relation
From “payments.entities.User”.

2.1.1.2) bind class “Model <Discount>”:
- Associates with Discount entity in a 1 to 1 relation
From “payments.entities.Discount”.

2.1.1.3) bind class “Model <FormElement>”:
- Associates with FormElement entity in a 1 to 1 relation
From “payments.entities.FormElement”.

2.1.1.4) bind class “Model <FormElementChoice>”:
- Associates with FormElementChoice entity in a 1 to 1 relation
From “payments.entities.FormElementChoice”.

2.1.1.5) bind class “Model <Provider>”:
- Associates with Provider entity in a 1 to 1 relation
From “payments.entities.Provider”.

2.1.1.6) bind class “Model <RefundRequest>”:
- Associates with RefundRequest entity in a 1 to 1 relation
From “payments.entities.RefundRequest”.

2.1.1.7) bind class “Model <Service>”:
- Associates with Service entity in a 1 to 1 relation
From “payments.entities.Service”.

2.1.1.8) bind class “Model <Transaction>”:
- Associates with Transaction entity in a 1 to 1 relation
From “payments.entities.Transaction”.

2.1.1.9) bind class “Model <UsedDiscount>”:
- Associates with UsedDiscount entity in a 1 to 1 relation
From “payments.entities.UsedDiscount”.

2.1.2) entities package:

2.1.2.1) Discount entity :
- Associates with “DiscountType" enum in a 1 to 1 relation
From “payments.enums.DiscountType “.

2.1.2.2) FormElement entity associates with FormElementType enum in a 1 to 1 relation
From “payments.enums.FormElementType”.

2.1.2.3) Provider entity associates with HandlerName enum in a 1 to 1 relation
From “handlers.HandlerName”.

2.1.2.4) Transaction entity associates with TransactionType enum in a 1 to 1 relation
From “payments.enums.TransactionType”.

2.1.3) controllers package:

2.1.3.1) Admin package:

2.1.3.1.1) AdminDiscountController class from “payments.controllers.admin”:
- Aggregates with the binding model class “Model <Discount>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with the entity class “Discount” in a 1 to many relation
From “payments.entities.Discount”
- Associates with the enum class “DiscountType” in a 1 to 1 relation
From “payments.enums.DiscountType.

2.1.3.1.2) AdminRefundController class from “payments.controllers.admin”:
- Aggregates with the binding model class “Model <RefundRequest>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “RefundRequest” entity in a 1 to many relation
From “payments.entities.RefundRequest”.
- Associates with “Transaction” entity in a 1 to 1 relation
from “payments.entities.Transaction”.
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.
- Associates with “RefundRequestStatus” enum in a 1 to 1 relation
From “payments.enums.RefundRequestStatus”.
- Associates with “TransactionType” enum in a 1 to 1 relation
From “payments.enums.TransactionType”.

2.1.3.1.3) AdminServiceController class from “payments.controllers.admin”:
- Aggregates with the binding model class “Model <Service>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <Provider>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <FormElement>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <FormElementChoice>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Service” entity in a 1 to 1 relation
From “payments.entities.Service”.
- Associates with “Provider” entity in a 1 to 1 relation
From “payments.entities.Provider”.
- Associates with “FormElement” entity in a 1 to 1 relation
From “payments.entities.FormeElement”.
- Associates with “FormElementChoice” entity in a 1 to 1 relation
From “payments.entities.FormElementChoice”.

2.1.3.1.4) AdminTransactionController class from “payments.controllers.admin”:
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Transaction” entity in a 1 to many relation
From “payments.entities.Transaction”.

2.1.3.1.5) AdminUserController class from “payments.controllers.admin”:
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation

From “datastore.Model”.
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.

2.1.3.2) Payment_Strategies package:
2.1.3.2.1) PayWithWallet class from “payments.controllers.Payment_Strategies”:
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.

2.1.3.3) request package:
2.1.3.3.1) AddFormElementBody class from “payments.controllers.request”:
- Associates with “FormElementType” enum in a 1 to 1 relation
From “payments.enums.FormElementType”.
1.3.3.2) AddProviderBody class from “payments.controllers.request”:
- Associates with “HandlerName” enum in a 1 to 1 relation
From “handlers.HandlerName”.

2.1.3.4) response package:
2.1.3.4.1) RefundRequestResponse class from “payments.controllers.response”:
- Associates with “RefundRequest” entity in a 1 to 1 relation
From “payments.entities.RefundRequest”.
- Associates with “Transaction” entity in a 1 to 1 relation
From “payments.entities.Transaction”.
- Associates with “RefundRequestStatus” enum in a 1 to 1 relation
From “payments.enums.RefundRequestStatus”.
- Associates with “TransactionType” enum in a 1 to 1 relation
From “payments.enums.TransactionType”.
2.1.3.4.2) UserResponse class from “payments.controllers.response”:
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.
2.1.3.4.3) HandlerResponse class from “payments.controllers.response”:
- Associates with “Handler” class in a 1 to 1 relation
From “handlers.Handler”.

2.1.3.5) AuthController class from “payments.controllers”:
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.

2.1.3.6) DiscountController class from “payments.controllers”:
- Aggregates with the binding model class “Model <Discount>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <UsedDiscount>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Discount” entity in a 1 to many relation

From “payment.entities.Discount”.
- Associates with “UsedDiscount” entity in a 1 to many relation
From “payment.entities.UsedDiscount”.
- Associates with “DiscountType” enum in a 1 to 1 relation
From “payments.enums.DiscountType”.

2.1.3.7) PaymentController class from “payments.controllers”:
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Discount” entity in a 1 to many relation
From “payments.entities.Discount”.
- Associates with “Provider” entity in a 1 to 1 relation
From “payments.entities.Provider”.
- Associates with “Transaction” entity in a 1 to 1 relation
From “payments.entities.Transaction”.
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.
- Associates with “DiscountType” enum in a 1 to 1 relation
From “payments.enums.DiscountType”.
- Associates with “TransactionType” enum in a 1 to 1 relation
From “payments.enums.TransactionType”.
- Associates with “Handler” in a 1 to 1 relation
From “handlers.Handler”.
- Associates with “HandlerFactory” in a 1 to 1 relation
From “handlers.HandlerFactory”.
- Associates with “HandlerResponse” in a 1 to 1 relation
From “handlers.HandlerResponse”.

2.1.3.8) ProviderController class from “payments.controllers”:
- Aggregates with the binding model class “Model <Provider>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Provider” entity in a 1 to many relation
From “payments.entities.Provider”.

2.1.3.9) RefundController class from “payments.controllers”:
- Aggregates with the binding model class “Model <RefundRequest>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “RefundRequest” entity in a 1 to many relation
From “payments.entities.RefundRequest”.
- Associates with “Transaction” entity in a 1 to 1 relation
From “payments.entities.Transaction”.

2.1.3.10) ServiceController class from “payments.controllers”:
- Aggregates with the binding model class “Model <Service>” in a 1 to 1 relation

From “datastore.Model”.
- Aggregates with the binding model class “Model <Provider>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <Discount>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <FormElement>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <FormElementChoice>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Service” entity in a 1 to many relation
From “payments.entities.Service”.
- Associates with “Provider” entity in a 1 to many relation
From “payments.entities.Provider”.
- Associates with “Discount” entity in a 1 to many relation
From “payments.entities.Discount”.
- Associates with “FormElement” entity in a 1 to many relation
From “payments.entities.FormElement”.
- Associates with “FormElementChoice” entity in a 1 to many relation
From “payments.entitiesFromElementChoice”.
- Associates with “FormElementType” enum in a 1 to 1 relation
From “payments.enums.FormElementType”.
- Associates with “HandlerFactory” in a 1 to 1 relation
From “handlers.HandlerFactory”.
- Associates with “Handler” in a 1 to 1 relation
From “handlers.Handler”.

2.1.3.11) TransactionController class from “payments.controllers”:
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <RefundRequest>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “Transaction” entity in a 1 to many relation
From “payments.entities.Transaction”.
- Associates with “RefundRequest” entity in a 1 to 1 relation
From “payments.entities.RefundRequest”.
- Associates with “TransactionType” enum in a 1 to 1 relation
From “payments.enums.TransactionType”.
- Associates with “RefundRequestStatus” enum in a 1 to 1 relation
From “payments.enums.RefundRequestStatus”.

2.1.3.12) UserController class from “payments.controllers”:
- Aggregates with the binding model class “Model <User>” in a 1 to 1 relation
From “datastore.Model”.
- Aggregates with the binding model class “Model <Transaction>” in a 1 to 1 relation
From “datastore.Model”.
- Associates with “User” entity in a 1 to 1 relation
From “payments.entities.User”.

- Associates with “Transaction” entity in a 1 to 1 relation
From “payments.entities.Transaction”.
- Associates with “TransactionType” enum in a 1 to 1 relation
From “payments.enums.TransactionType”.

2.3 handler package

- All relations reaching into the handlers package have been mapped in section 1.
- No relations are reaching out from the handlers package to any other package in the system.

Relevant Association and multiplicity mapping within this view :

2.3.1) Handler (abstract class):
- Associates with HandlerResponse class in a 1 to 1 relation.
- Associates with HandlerName enum in a 1 to 1 relation.

2.3.2) HandlerFactory :
- Aggregates with the Handler class in a 1 to many relation.
- Associates with HandlerName enum in a 1 to 1 relation.
- Associates with “CancerHospitalDonationHandler” class in a 1 to 1 relation
From “handlers.concrete.CancerHospitalDonationHandler”.
- Associates with “ErroneousHandler” class in a 1 to 1 relation
From “handlers.concrete.ErroneousHandler”.
- Associates with “EtisalatInternetHandler” class in a 1 to 1 relation
From “handlers.concrete.EtisalatInternetHandler”.
- Associates with “EtisalatRechargeHandler” class in a 1 to 1 relation
From “handlers.concrete.EtisalatRechargeHandler”.
- Associates with “MonthlyLandlineHandler” class in a 1 to 1 relation
From “handlers.concrete.MonthlyLandlineHandler”.
- Associates with “NgoDonationHandler” class in a 1 to 1 relation
From “handlers.concrete.NgoDonationHandler”.
- Associates with “OrangeInternetHandler” class in a 1 to 1 relation
From “handlers.concrete.OrangeInternetHandler”.
- Associates with “OrangeRechargeHandler” class in a 1 to 1 relation
From “handlers.concrete.OrangeRechargeHandler”.
- Associates with “QuarterlyLandlineHandler” class in a 1 to 1 relation
From “handlers.concrete.QuarterlyLandlineHandlerr”.
- Associates with “SchoolDonationHandler” class in a 1 to 1 relation
From “handlers.concrete.SchoolDonationHandler”.
- Associates with “VodafoneInternetHandler” class in a 1 to 1 relation
From “handlers.concrete.VodafoneInternetHandler”.
- Associates with “VodafoneRechargeHandler” class in a 1 to 1 relation
From “handlers.concrete.VodafoneRechargeHandler”.
- Associates with “WeInternetHandler” class in a 1 to 1 relation
From “handlers.concrete.WeInternetHandler”.
- Associates with “WeRechargeHandler” class in a 1 to 1 relation
From “handlers.concrete.WeRechargeHandler”.

2.3.3) HandlerResponse doesn’t have any associations reaching out of it.
2.3.4) HandlerName enum can’t have any associations reaching out of it.

2.3.5) Concrete Handlers Package :
“CancerHospitalDonationHandler”, “ErroneousHandler”, “EtisalatInternetHandler”, “EtisalatRechargeHandler”, “MonthlyLandlineHandler”, “NgoDonationHandler”, “OrangeInternetHandler”, “OrangeRechargeHandler”, “QuarterlyLandlineHandler”, “SchoolDonationHandler”, “VodafoneInternetHandler”, “VodafoneRechargeHandler”, “WeInternetHandler”, “WeRechargeHandler” from “handlers.concretes” all:
- extend the Handler class.
- Associate with HandlerName enum in a 1 to 1 relation.
- Associate with HandlerResponse class in a 1 to 1 relation.

2.4 payments.entities package
- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- No relations reaching into this package.

2.5 payments.enums package
- No relations are present within this view.
- No relations can reach out from this view.
- Any relations reaching into the Enumerations package have been mapped in section 1.

2.6 payments.controllers package
- All relations are completely mapped and are clear in this view except for the following :
2.6.1) Admin package :

2.6.1.1) AdminDiscountController in “payments.controllers.admin”:
- Aggregates with “ServiceController” in a 1 to 1 relation
From “payments.controllers.ServiceController”.
- Associates with “AddDiscountBody” in a 1 to 1 relation
From “payments.controllers.request.AddDiscountBody”.

2.6.1.2) AdminRefundController in “payments.controllers.admin”:
- Associates with “RefundRequestResponse” in a 1 to 1 relation
From “payments.controllers.response.RefundRequestResponse”.

2.6.1.3) AdminRefundController in “payments.controllers.admin”:
- Aggregates with “ServiceController” in a 1 to 1 relation
From “payments.controllers.ServiceController”.
- Associates with “AddFormElementBody” in a 1 to 1 relation
From “payments.controllers.request.AddFormElementBody”.
- Associates with “AddFormElementChoiceBody” in a 1 to 1 relation
From “payments.controllers.request.AddFormElementChoiceBody”.
- Associates with “AddProviderBody” in a 1 to 1 relation
From “payments.controllers.request.AddProviderBody”.
- Associates with “AddServiceBody” in a 1 to 1 relation
From “payments.controllers.request.AddServiceBody”.

2.6.1.4) AdminUserController in “payments.controllers.admin”:
- Associates with “UserResponse” in a 1 to 1 relation
From “payments.controllers.response.UserResponse”.

2.7 payments.controllers.payment_strategies package
- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- Any relations reaching into this package have been mapped in previous sections.

2.8 payments.controllers.response
- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- Any relations reaching into this package have been mapped in previous sections.

2.9 payments.controllers.request
- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- Any relations reaching into this package have been mapped in previous sections.

2.10 payments.controllers.admin package
- All relations within this view are completely mapped in the class diagram.
- All relations reaching out from this package to any other package were mapped in previous sections.
- No relations reaching into this package.

3. Design Patterns

3.1. Template Method

All of the handlers need to check if the request contains the required keys (check section 1.5) and all of them need to handle the request and extract the amount to deduct. However, each one provides its own implementation for getting the required keys to check against and for handling the request and extracting the amount. handlers.Handler is what forces this template, and classes in the handlers.concretes package implement it.

3.2. Simple Factory

The providers, which are dynamic entities, contain the handler name. But the handler is a static asset in the code. So to be able to choose the appropriate handler for the provider we have a handlers.HandlerFactory which uses the simple factory pattern. Its intent is to return the appropriate instance of the handler needed by the controllers using the handlerName in the provider object.

3.3. Strategy

We need to select entities from the model based on different conditions, hence datastore.Model uses the strategy pattern for the selection and update logic in methods like select(), selectMax(), update(). These methods take an object implementing the strategy interface. In the code, this interface is typically implemented using lambda expressions in the controllers. An example of the strategy interface is RecordFilterStrategy; the strategy function (apply) in this interface takes an object and returns a boolean if it satisfies the filter condition. This allows us to use the Model generically.

payments.controllers.payment_strategies package uses the strategy pattern for the selection of the payment strategy and each strategy implements a different payment method according to type. This appears in the interface method pay() to be implemented by all concrete strategies. This decouples the controllers from the implementations of different strategies. An example is in PayWithCreditCard; the strategy function (pay) in this concrete class does all the appropriate logical checks for payment with credit card while the controller invoking it doesn't know which concrete strategy is being executed.

3.4. Decorator

The decorator pattern is intentionally not used. One might think it would be a good fit for accumulating the discounts; however, we disagree with this point of view. The intent of the decorator pattern is to allow behavior to be added to individual objects. So we have a base object and augmented objects that add the behavior. This does not apply to discounts. All discounts are similar (whether overall or specific). There is no notion of a base object and augmented objects.

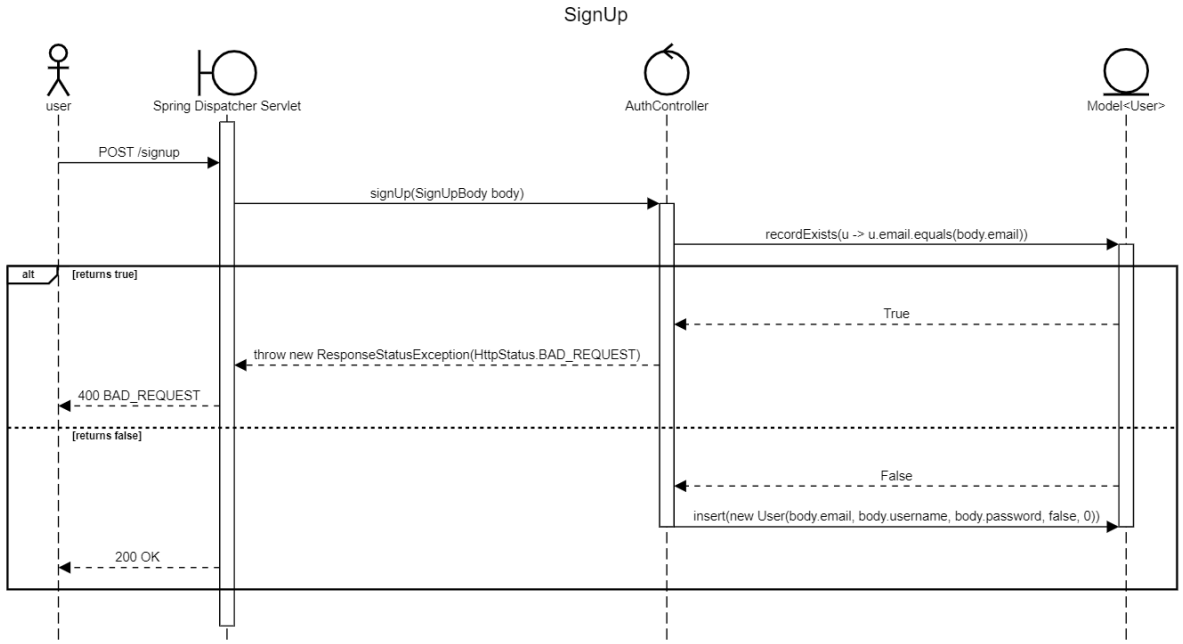
3.5. Abstract factory

The abstract factory pattern is intentionally not used. One might think it would be a good fit for services and service providers; however, we disagree with this point of view. The intent of the abstract factory is to encapsulate a group of factories that **have a common theme**. There is no need to shoehorn services and service providers into factories and products when we can simply represent them as a one-to-many relationship in our relational model, and use the simple factory pattern to resolve the handler as discussed in 3.2.

4. Sequence Diagrams

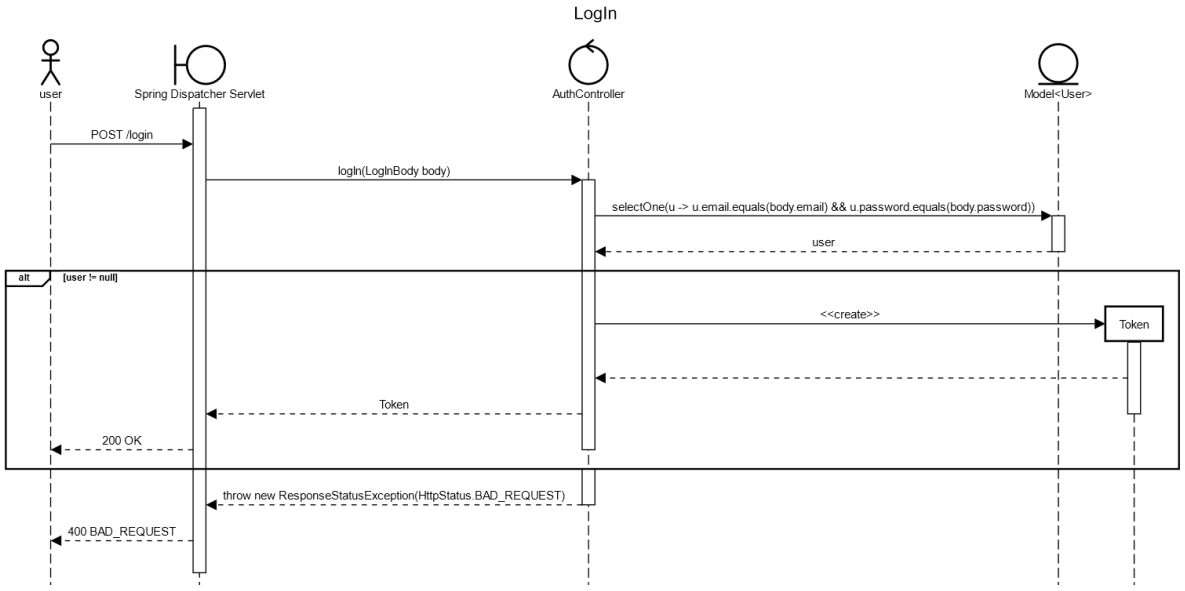
4.1. SignUp

The user attempts to create an account with the ‘POST/signup’ endpoint



4.2. Login

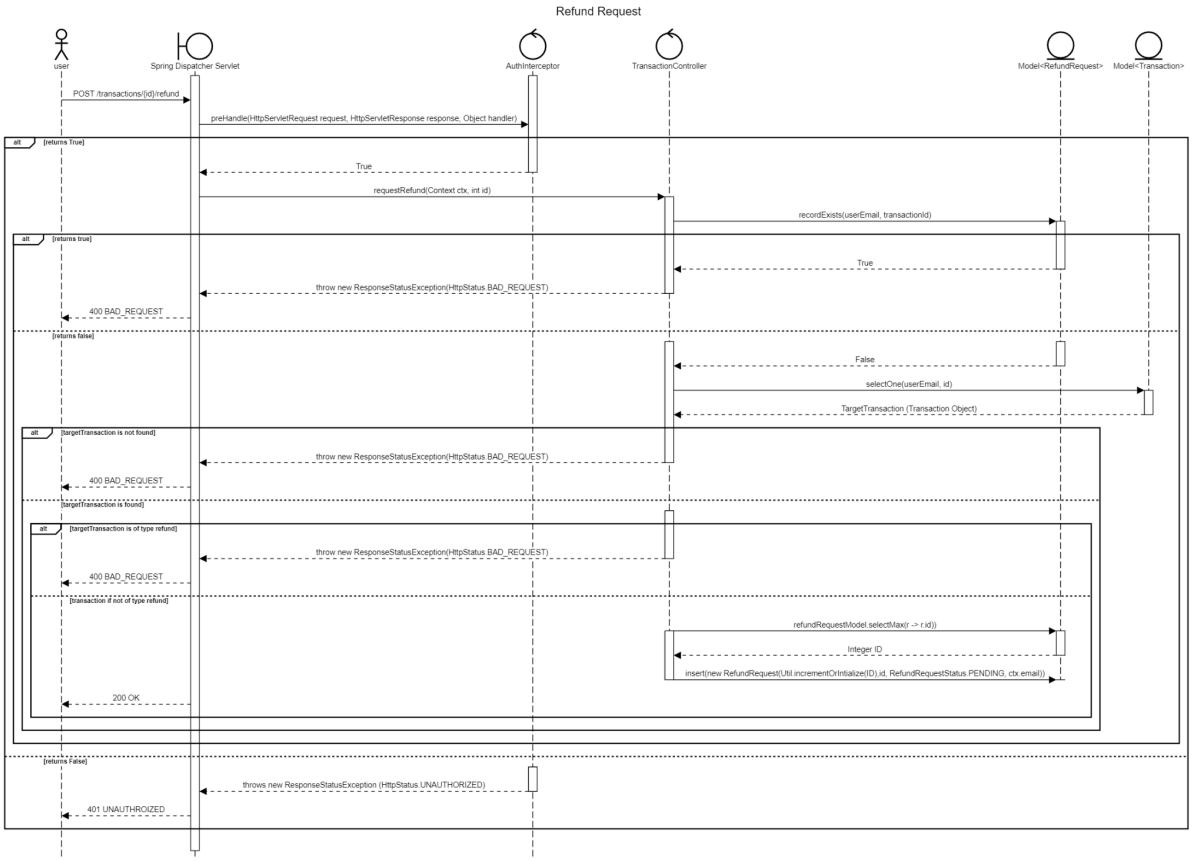
The user uses the “POST /login” endpoint to authenticate with the system. LogInBody object is retrieved from the request, email and password and compared with the stored versions. If a match is found, a token object is created and given to the user to authenticate them and a 200 OK is returned with the token; otherwise a 400 BAD_REQUEST is returned.



4.3. Refund request

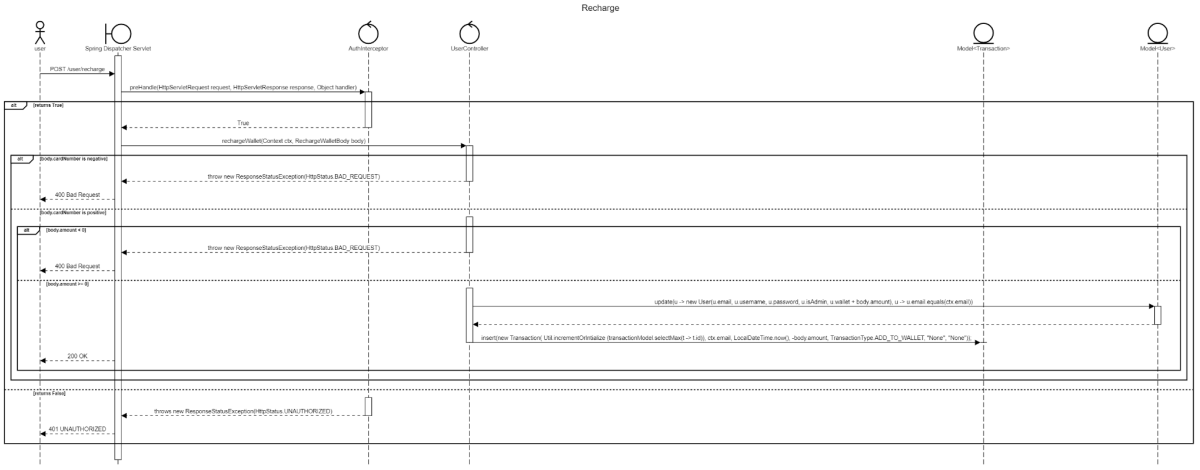
The user uses the “POST /transactions{id}/refund” endpoint to post a refund request on a transaction. The user must be logged in, not have an existing refund request on the target transaction. The target transaction must not be of type “REFUND”. If all conditions are met, then the system will create a refund object with appropriate user details on the target transaction with pending status. This object will be inserted into the Refund Request model and a 200 OK is returned.

Failure to authenticate returns 401 UNAUTHORIZED.



4.4. Recharge

- User attempts to recharge his wallet using the “POST /user/recharge” endpoint. User first authenticates, failure to authenticate will return 401 UNAUTHORIZED. The UserController receives the RechargeWalletBody from the request and extracts the credit card number and amount to validate them. If any of them are invalid, 400 BAD_REQUEST is returned. Otherwise a user account is updated with the new balance in the UserModel and a new Transaction object is inserted into the TransactionModel with the details of the transaction. 200 OK is returned in case of success.



4.5. Pay cash (better viewed in docs/sequence-diagram)

- The user attempts to pay for a service using cash via the “POST /services/{serviceName}/providers/{providerName}/pay-cash” endpoint.

If failed to Authenticate, 401 UNAUTHORIZED is returned and sequence terminated.

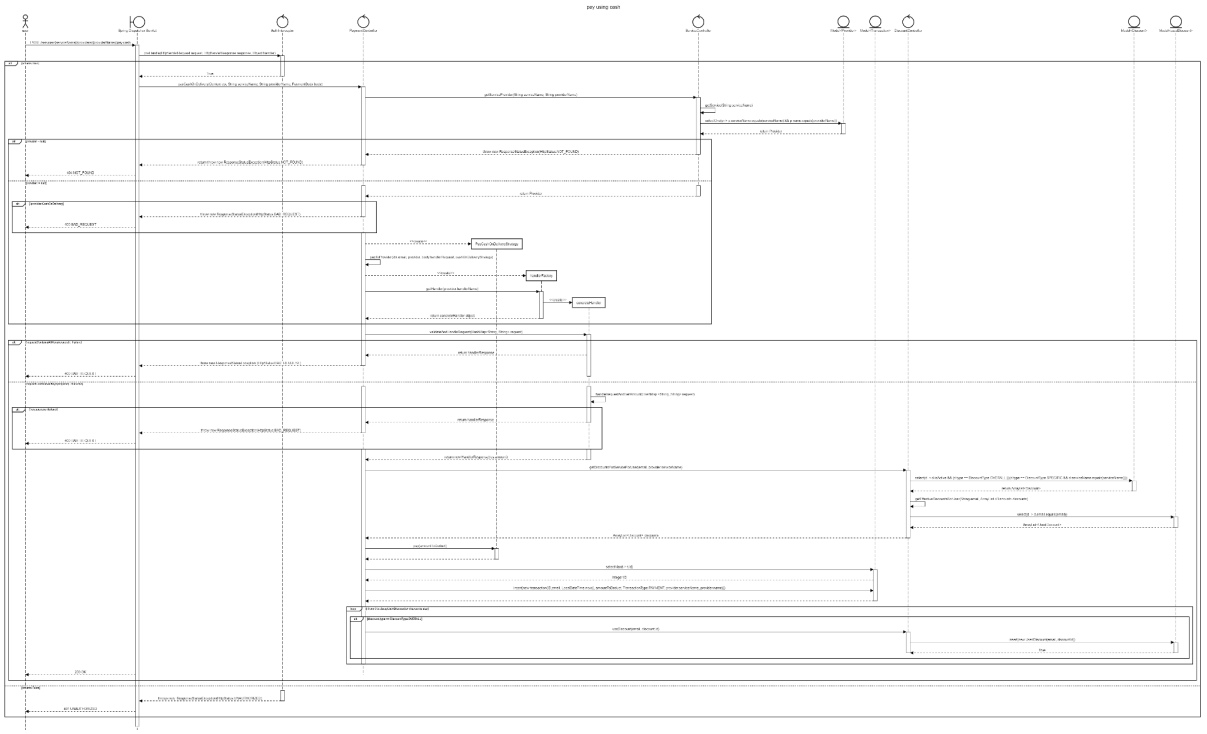
If the provider returned is not found then “404 NOT_FOUND” is returned.

If the provider returned does not support cash on delivery then “400 BAD_REQUEST” is returned.

If the request does not contain all keys a handlerResponse object is returned and “400 BAD_REQUEST” is displayed.

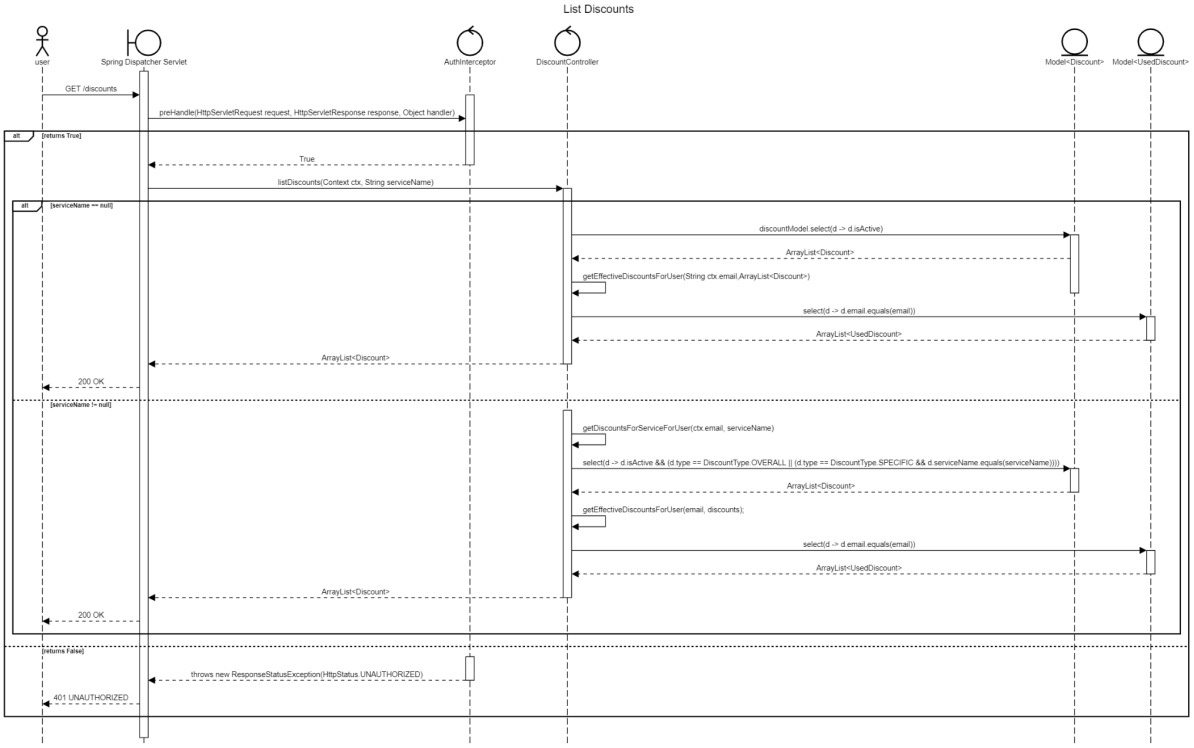
If there is failure in processing the request then “400 BAD_REQUEST” is returned as a response with appropriate message.

If all of the above cases are passed successfully, 200 OK is returned and the transaction processes the changes in the usedDiscount Model and transactionModel.



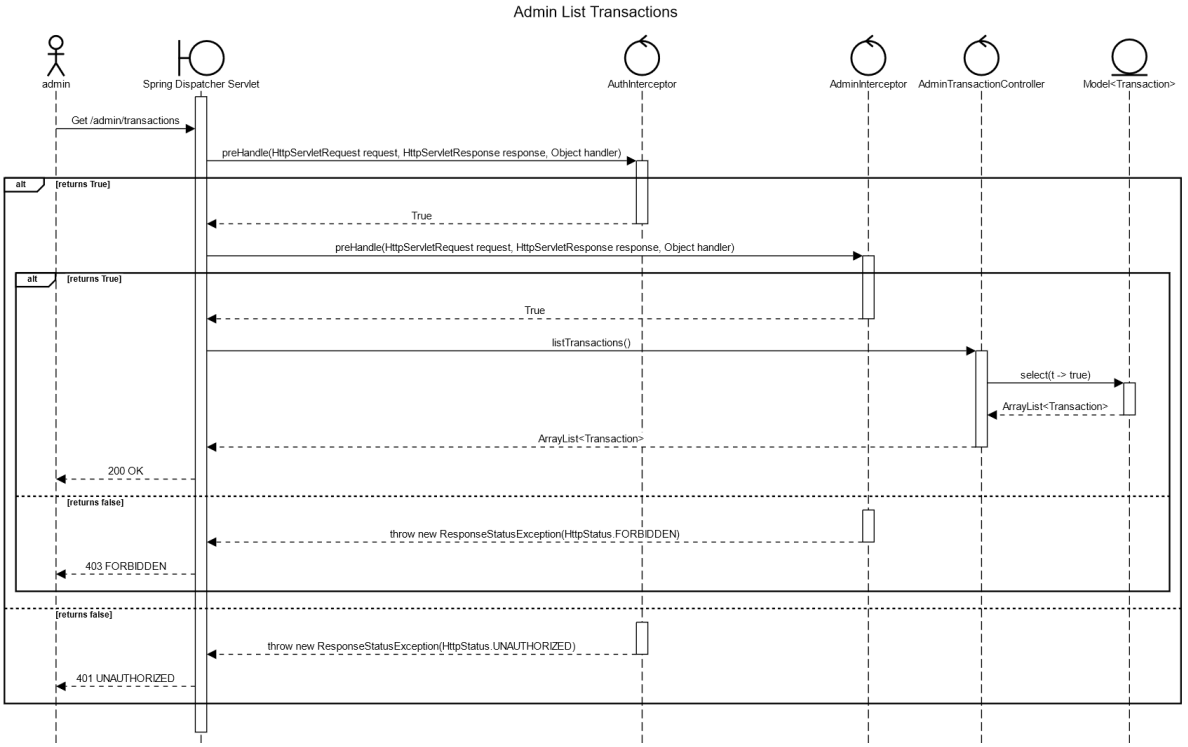
4.6. List Discounts

- The User attempts to list either all the discounts that are eligible to the user in the system or eligible discounts for a specific service by entering a service name. This is done using the “GET/discounts” endpoint. The user authenticates, if fails a “401 UNAUTHORIZED” is returned.When “listDiscounts(context,serviceName)” method is called and no service name entered, all the possible discounts and a “200 OK ” is returned. If a service name was entered, discounts are filtered with method “”getDiscountsForServiceForUser(string, ArrayList <Discount>)” and then effective discounts are retrieved by comparing against UsedDiscounts array from the UsedDiscounts Model. At the end the array list of discounts and “200 OK” are returned.



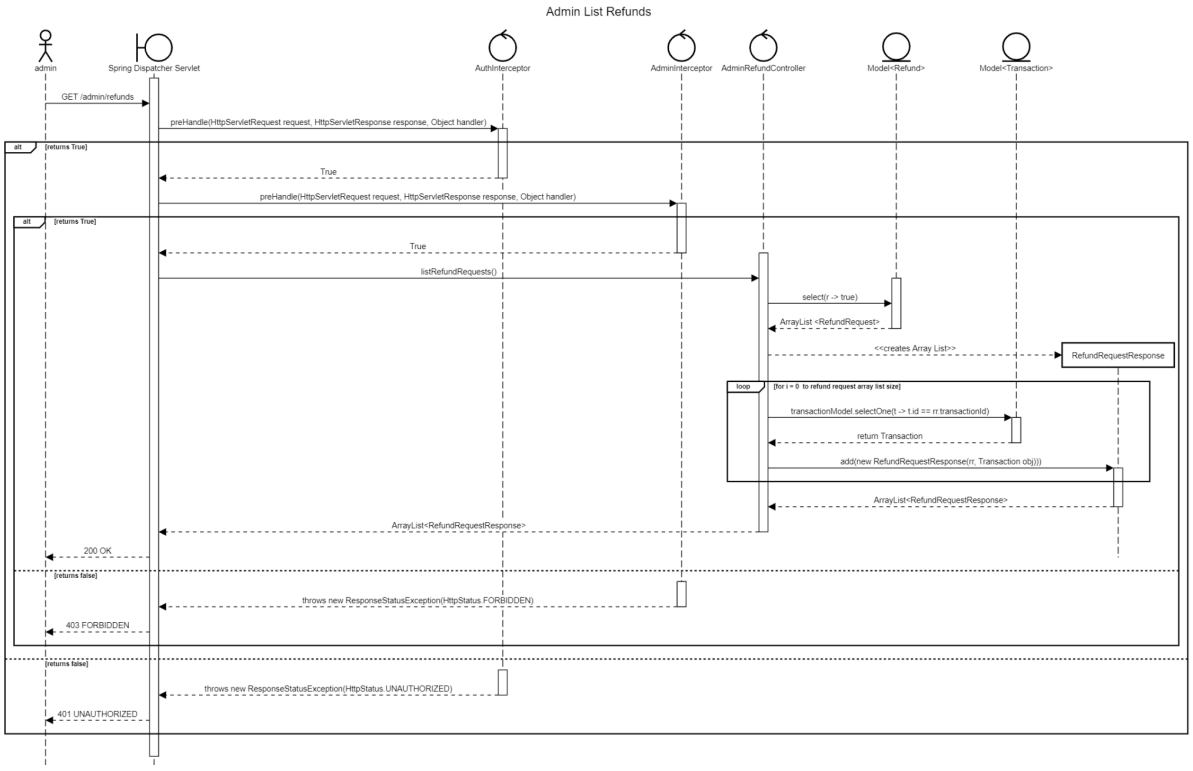
4.7. Admin list transactions

- An Admin attempts to list all system transactions through the “Get /admin/transactions” endpoint. The sender is first to be authenticated as the user via the AuthInterceptor. If failed, return “401 UNAUTHORIZED”. Afterwards the user is authenticated for admin privileges via the AdminInterceptor. If failed, return “403 FORBIDDEN”. At this point the sender is known to be an admin so, the “listTransactions()” in AdminTransactionController is invoked which in turn fetches all the transactions from the transactionModel via transactionModel.select(t -> true). An arraylist of transactions and a “200 OK” is returned in this case.



4.8. Admin list refunds:

- An Admin attempts to list all system Refund Requests through the “Get /admin/refunds” endpoint. The sender is first to be authenticated as the user via the AuthInterceptor. If failed, return “401 UNAUTHORIZED”. Afterwards the user is authenticated for admin privileges via the AdminInterceptor. If failed, return “403 FORBIDDEN”.At this point the sender is known to be an admin so, the “listRefundRequests()” in AdminRefundController is invoked which in turn fetches an arraylist of all Refund Requests from the RefundRequest Model. Afterwards, an arraylist of RefundRequestResponse is created and each object of this array will hold the refund request and the associated transaction. The transactions are fetched from the transactionModel in a loop through matching their id with the transaction id stored in the RefundRequests arraylist using the “transactionModel.selectOne(t -> t.id == rr.transactionId)”. At the end the RefundRequestResponse arraylist containing all the info and a “200 OK” is returned.



5. API endpoints

The Postman collection can be imported from docs/postman/collection.json. The collection has a flow-of-events folder. Running all of the requests in this folder in sequence simulates a normal flow of events with an admin and a user.

For every endpoint, the API returns the appropriate status code and error response if an error happens. The appropriate auth header must be provided for controllers that require authorization/authentication. If you are using our Postman collection, this header is set automatically on login.

In the following documentation, if a response is "N/A," this means the response only contains the status and headers. If a request body is "N/A," this means only the appropriate headers are required.

5.1. User

User: {
  "email": string,
  "username": string,
  "isAdmin": boolean,
  "wallet": double
}

5.1.1. Signup

Endpoint: POST /signup
Body: {
  "username": string,
  "email": string,
  "password": string
}
Response: N/A
Requirement: The user should be able to sign up to the system

5.1.2. Login

Endpoint: POST /login
Body: {
  "email": string,
  "password": string
}
Response: {
  "token": string
}
Requirement: The user should be able to sign-in to the system

5.1.3. GET (own profile)

Endpoint: GET /user
Body: N/A
Response: User
Requirement: the user should be able to view his own profile (inferred)

5.1.4. Recharge (wallet)
Endpoint: POST /user/recharge
Body: {
  "amount": number,
  "cardNumber": string
}
Response: N/A
Requirement: the user should be able to add any funds to the wallet

5.1.5. List
Endpoint: GET /admin/users
Body: N/A
Response: [User, …]
Requirement: the admin should be able to list all users (extra)

5.1.6. Get with email (any user)
Endpoint: GET /admin/users/{email}
Body: N/A
Response: User
Requirement: the admin should be able to get a certain user (extra)

5.1.7. Check (won't be implemented)
Endpoint: GET /users/check (example in instructions)
Explanation: this example endpoint unnecessarily exposes a detail in a controller action (checking if a user exists). It does nothing more than a signup dry run. For this purpose, you can either use the /signup endpoint (to actually sign up if possible) or the /admin/users/{id} (to get a user by their id or return 404 if the user is not found)

5.2. Service
Service: {
  "name": string
}

5.2.1. List
Endpoint: GET /services
Body: N/A
Response: [Service, …]
Requirement: the user should be able to list all of the services (inferred)

5.2.2. Get one
Endpoint: GET /services/{serviceName}
Body: N/A
Response: Service
Requirement: The user should be able to search for any service in the system

5.2.3. Search
Endpoint: GET /providers?name={partial_or_full_service_or_provider_name}

Body: N/A
Response: [Provider, …] // Providers will be discussed in (5.3)
Requirement: The user should be able to search for any service in the system

5.2.4. Add
Endpoint: POST /admin/services
Body: {
  "name": string
}
Response: N/A
Requirement: The admin should be able to add any service to the system (inferred)

5.3. Provider
Provider: {
  "serviceName": string,
  "name": string,
  "cashOnDelivery": boolean,
  "handlerName": string
}

5.3.1. List all
Endpoint: GET /providers
Body: N/A
Response: [Provider, …]
Requirement: the user should be able to list all of the providers in the system (inferred)

5.3.2. Search
Endpoint: GET /providers?name={partial_or_full_service_or_provider_name}
Body: N/A
Response: [Provider, …]
Requirement: the user should be able to search for any provider in the system (inferred)

5.3.3. List for a service
Endpoint: GET /services/{serviceName}/providers
Body: N/A
Response: [Provider, …]
Requirement: the user should be able to list all of the providers providing a service (inferred)

5.3.4. Get one
Endpoint: GET /services/{serviceName}/providers/{providerName}
Body: N/A
Response: Provider
Requirement: the user should be able to get a provider by its name and its service name (extra)

5.3.5. Pay wallet (if sufficient)
Endpoint: POST /services/{serviceName}/providers/{providerName}/pay-wallet

Body: {
  "handlerRequest": {
    string: string,
    …
  }
}
(handler requests are discussed in section 1.5)
Response: N/A
Requirement: the user can pay for any service in the system (wallet)

5.3.5. Pay cash (if possible)
Endpoint: POST /services/{serviceName}/providers/{providerName}/pay-cash
Body: {
  "handlerRequest": {
    string: string,
    …
  }
}
(handler requests are discussed in section 1.5)
Response: N/A
Requirement: the user can pay for any service in the system (cash)

5.3.7. Pay credit card
Endpoint: POST /services/{serviceName}/providers/{providerName}/pay-credit-card
Body: {
  "handlerRequest": {
    string: string,
    …
  }
}
(handler requests are discussed in section 1.5)
Response: N/A
Requirement: the user can pay for any service in the system (credit card)

5.3.8. Get handler
Endpoint: GET /services/{serviceName}/providers/{providerName}/handler
Body: N/A
Response: {
  "requiredRequestAttributes": [
    string, …
  ],
  "constraints": string
}
(discussed in section 1.5)
Requirement: system specific

5.3.9. Add
Endpoint: POST /admin/services/{serviceName}/providers

Body: {
"name": string
"cashOnDelivery": boolean,
"handlerName": string
}
Response: N/A
Requirement: the admin should be able to add any new service provider to the system

5.4. Form elements
FormElement: {
"name": string,
"serviceName": string,
"providerName": string,
"type": string,
"info": string
}

5.4.1. List for service provider
Endpoint: GET /services/{serviceName}/providers/{providerName}/form-elements
Body: N/A
Response: [FormElement, …]
Requirement: provider consists of a form to be sent to the user

5.4.2. Get
Endpoint: GET
/services/{serviceName}/providers/{providerName}/form-elements/{formElementName}
Body: N/A
Response: FormElement
Requirement: provider consists of a form to be sent to the user

5.4.3. Add
Endpoint: POST /admin/services/{serviceName}/providers/{providerName}/form-elements
Body: {
"name": string
"info": string
"type": string
}
Response: N/A
Requirement: provider consists of a form to be sent to the user

5.5. Form element choices
FormElementChoice: {
"info": string,
"formElementName": string,
"serviceName": string,
"providerName": string
}

5.5.1. Get (if dropdown)
Endpoint: GET /services/{serviceName}/providers/{providerName}/form-elements/{formElementName}/choices
Body: N/A
Response: [FormElementChoice, …]
Requirement: provider consists of a form to be sent to the user

5.5.2. Add
Endpoint: POST /admin/services/{serviceName}/providers/{providerName}/form-elements/{formElementName}/choices
Body: {
"info": string
}
Response: N/A
Requirement: provider consists of a form to be sent to the user

5.6. Discounts
Discount: {
"id": int,
"type": string,
"percentage": double,
"serviceName": string,
"isActive": boolean
}

5.6.1. List (only ones applicable to the user)
Endpoint: GET /discounts
Body: N/A
Response: [Discount, …]
Requirement: the user should be able to list all discounts in the system applicable to him (inferred)

5.6.2. List for service (only ones applicable to the user)
Endpoint: GET /services/{serviceName}/discounts
Body: N/A
Response: [Discount, …]
Requirement: the user should be able to check any discount for any service in the system [that are applicable to him]

5.6.3. List all
Endpoint: GET /admin/discounts
Body: N/A
Response: [Discount, …]
Requirement: the admin should be able to list all discounts in the system (inferred)

5.6.4. Add overall

Endpoint: POST /admin/discounts
Body: {
  "percentage": 30
}
Response: N/A
Requirement: the admin should be able to add discounts to the system (overall)

5.6.5. Add specific
Endpoint: POST /admin/services/{serviceName}/discounts
Body: {
  "percentage": 30
}
Response: N/A
Requirement: the admin should be able to add discounts to the system (specific)

5.6.6. Deactivate
Endpoint: POST /admin/discounts/{id}/deactivate
Body: N/A
Response: N/A
Requirement: the admin should be able to deactivate any discount (inferred)

5.6.7. Activate
Endpoint: POST /admin/discounts/{id}/activate
Body: N/A
Response: N/A
Requirement: the admin should be able to activate deactivated discounts (inferred)

5.7. Refunds
Refund: {
  "id": number,
  "status": string,
  "serviceName": string,
  "providerName": string,
  "amount": double,
  "userEmail": string,
  "timeStamp": string,
  "type": string
}

5.7.1. List (user's refunds)
Endpoint: GET /refunds
Body: N/A
Response: [Refund, …]
Requirement: the user should be able to list his own refunds (inferred)

5.7.2. Request
Endpoint: POST /transactions/{id}/refund
Body: N/A

Response: N/A
Requirement: the user can ask for a refund for any complete transaction to any given service

5.7.3. List all
Endpoint: GET /admin/refunds
Body: N/A
Response: [Refund, …]
Requirement: the admin should be able to list all refund requests

5.7.4. Get
Endpoint: GET /admin/refunds/{id}
Body: N/A
Response: Refund
Requirement: the admin should be able to get a refund with a certain id (inferred)

5.7.5. Accept
Endpoint: POST /admin/refunds/{id}/accept
Body: N/A
Response: N/A
Requirement: the admin should be able to accept or reject any refund request

5.7.6. Reject
Endpoint: POST /admin/refunds/{id}/reject
Body: N/A
Response: N/A
Requirement: the admin should be able to accept or reject any refund request

5.8. Transactions
Transaction: {
```
  "id": number,
  "userEmail": string,
  "timestamp": string,
  "amount": double,
  "type": string,
  "serviceName": string,
  "providerName": string
}
```

5.8.1. List (own transactions)
Endpoint: GET /transactions
Body: N/A
Response: [Transaction, …]
Requirement: the user should be able to list his own transactions (inferred)

5.8.2. Get (own transaction)
Endpoint: GET /transactions/{id}
Body: N/A

Response: Transaction
Requirement: the user should be able to get one of his transactions using its id (inferred)

5.8.3. List all

Endpoint: GET /admin/transactions
Body: N/A
Response: [Transaction, …]
Requirement: the admin should be able to list all user transactions

6. GitHub repository

Link: https://github.com/sda-assignment/sda-assignment/

Invited TAs usernames:

- mhmdsamir92 (invitation accepted)
- HassanMouradMohamed (invitation accepted)
- esraasalemahmed (invation expired)
- a.samir@fci-cu.edu.eg (invitation pending)
- AhmedSamir17 (invitation pending)