Technical University of Denmark

Technical University of Denmark

02267 Software Development of Web Services

DTUPay - User's Guide

Tobias Rydberg (s173899)
Daniel Larsen (s151641)
Emil Kosiara (s174265)
Troels Lund (s161791)
Sebastian Lindhard Budsted (s135243)
Kasper Lindegaard Stilling (s141250)

Group 11

January 19, 2021

REST Interface overview

As a recap, the table of URIs from the report is included here.

URI Path = Resource	HTTP Method	Function
`/account`	POST	Register a user
`/account`	GET	Retrieves all users
`/account/{id}`	GET	Retrieve a user by ID
`/account/{id}`	DELETE	Retires a user by ID
`/account/by-cpr/{cpr}`	GET	Retrieve a user by CPR
`/account/by-cpr/{cpr}`	DELETE	Retires a user by CPR
`/payment`	POST	Perform a transaction
`/payment/refund`	POST	Performs a refund transaction
`/token`	POST	Requests tokens
`/token/{id}`	GET	Retrieves tokens of a user by ID
`/token/{id}`	DELETE	Retires tokens of a user by ID
`/report/customer/{id}`	GET	Requests transactions of a user **Note:** port 8083
`/report/merchant/{id}`	GET	Requests transactions of a merchant **Note:** port 8083
`/management`	GET	Request summary and transactions. **Note:** port 8083

Table 1: REST interface overview

Use cases

Each service and its possible URIs are described as subsections.

Account service

Registering a user is achieved through the REST interface at the `/account` path by issuing a POST request containing a user registration object. When this occurs, the REST service sends out an event to RabbitMQ with type `Register`. The account service is listening for this event and reacts when it appears by trying to register the account in cooperation with the provided bank service, using its SOAP interface. If the registration succeeds the account service sends a `RegisterSuccessful` event back to RabbitMQ. A `RegisterFailed` event is sent if registration fails. The REST service listens for both of the RabbitMQ events and returns HTTP 200 along with the ID if successful and HTTP 400 Bad Request otherwise.

Get account by ID is achieved through the REST interface at the `/account/{id}` path by issuing a GET request. The provided ID is the path parameter used to search for an account. The ID is then sent in an event of type `GetAccount` to RabbitMQ. The account service listens for this event and

tries to retrieve the account matching the specified ID from its repository of accounts. If it succeeds, an event of type `GetAccountSuccessful` is sent back to RabbitMQ containing the user account. An event of type `GetAccountFailed` is sent if something goes wrong e.g. when the user does not exist. The REST service listens for both events and returns HTTP 200 along with the account object if successful and HTTP 400 Bad Request otherwise.

Get account by CPR is very similar to the previous scenario but the REST path is located at `/account/by-cpr/{cpr}` and the path parameter is the CPR number to be searched for. Other than that, it follows the same chain of events as getting by ID, albeit with different message type names: (`GetAccountByCpr, GetAccountByCprSuccessful, GetAccountByCprFailed`)

Retire an account by ID is achieved through the REST interface at the `/account/{id}` path by issuing a DELETE request. The path parameter is the ID of the user to be deleted. When the service receives the REST call it sends an event of the type `RetireAccount`, along with the requested ID, to RabbitMQ. The account service is listening for this event and tries to retire the account associated with the specified ID. If it succeeds, an event of message type `RetireAccountSuccessful` is sent back to RabbitMQ. If it fails, an event of message type `RetireAccountFailed` is sent instead. The REST service is listening for both events and if the call was not successful, HTTP 400 Bad Request is returned. If it succeeds a status code 200 is returned to the requester.

Retire an account by CPR is very similar to the ID approach but the REST path is located at `/account/by-cpr/{cpr}` and the path parameter is the CPR number of the user to be deleted. Other than that, it follows the same chain of events as retiring by ID, albeit with different message type names: (`RetireAccountByCpr, RetireAccountByCprSuccessful, RetireAccountByCprFailed`)

Retrieving all accounts is achieved through the REST interface at the `/account` path by issuing a GET request. When this occurs, the REST service sends an event of message type `GetAllAccounts` to RabbitMQ which is an event that the account service is listening for. The account service then tries to get a list of all the users in its repository and puts that list inside an event of type `GetAllAccountsSuccessful` if nothing goes wrong and an event of type `GetAllAccountsFailed` if something does go wrong. The REST service is listening for these events and returns the list of users if the event succeeds, along with a status code 200, and HTTP 400 Bad Request is returned otherwise.

Payment service use cases

Paying an amount to a merchant is achieved through the REST interface at the `/payment` path by issuing a POST request containing a payment request object. This REST call triggers an event of type `ProcessPayment` which is sent to RabbitMQ. The account service is listening for events with this message type and it verifies the existence of the customer and merchant contained in the payment object and then packs a new object con-

taining the verified accounts, the amount and the token, into a new event with the message type `PaymentAccountsSuccessful` which is sent to RabbitMQ. The token service is listening for this event and when received it retrieves the token and tries validate it. If validation succeeds an event is sent to RabbitMQ with the message type `TokenValidationSuccessful` which is an event that the payment service is listening for. When the event occurs the payment service attempts to perform the transaction using the SOAP interface to the bank service. If the transaction succeeds, an event of message type `PaymentSuccessful` is sent to RabbitMQ which is an event that the REST service is listening for and finally the transaction details is recorded and a status code 200 is returned along with the transaction object. If either the verification of accounts or validation of the token fails, corresponding `PaymentAccountsFailed` and `TokenValidationFailed` -events will be sent to RabbitMQ which are picked up by the payment service. In both cases, a new event containing the transaction details and of message type `PaymentFailed` is sent to RabbitMQ. Finally, the REST service is listening for this event and records the transaction failure.

Refunding an amount to a customer is essentially identical to paying a merchant, only the REST path is located at `/payment/refund` but the subsequent events are actually payment events. If everything goes well a status code 200 is returned to the requester along with the transaction object.

Token service use cases

Requesting new tokens to user is achieved through the REST interface of the REST service at the `/token` path by issuing a POST request with the query parameters `id` and `amount`. The specified parameters is packed into an event of type `RequestTokens` which is sent to RabbitMQ. The token service is listening for this exact event and when received, it tries to create the specified amount of tokens for the specified user and then it sends an event back to RabbitMQ. If it succeeded, an event of message type `RequestTokensSuccessful` containing the customer ID is sent to RabbitMQ. If it failed, a similar event is sent but of type `RequestTokensFailed`. These events are listened for by the REST service and if it fails a corresponding exception is thrown. If it succeeds a status code 200 is returned.

Getting current tokens of user is achieved through the REST interface of the REST service at the `/token/{id}` path by issuing a GET request with the path parameter `id`, specifying the targeted user to get the tokens from. The resulting event from this REST call is of message type `GetToken` where the contents of the event is the ID of the user. The event is sent to RabbitMQ and the token service is listening for that exact event and if received, it tries to get the associated tokens from its repository and pack them in a new event. If it succeeds, an event of type `RequestTokensSuccessful` along with the token object is sent to RabbitMQ. If it fails, an event of type `RequestTokensFailed` is sent to RabbitMQ. The REST service is listening for these events and returns a status code corresponding to the events.

Retiring tokens is achieved through the REST interface of the REST service at the `/token/{id}` path by issuing a DELETE request, where the path parameter `id` is specifying the targeted user to retire the tokens from. The REST service then sends an event of type `RetireCustomerTokens` to RabbitMQ along with ID of the customer. This event is one that the token service is listening for and when received it tries to delete the tokens from the customer associated with the provided ID. If the operation is successful it sends an event to RabbitMQ of type `CustomerRetirementSuccessful` which is an event that the REST service is listening for. Finally a status code 200 is returned to the requester. If the operation is unsuccessful the token service sends and event of type `CustomerRetirementFailed` and the REST service pickes up the event and returns HTTP 400 Bad Request to the requester.

Reporting service use cases

Manager requesting summary and all transactions is achieved through a separate REST interface than the rest of the services, having a personal port accessible on 8083 for all REST API calls. The manager can on /management request for a summary consisting of the min, max, mean and sum of all the transactions in the system and can locally work with the data as well through the list of transactions being returned to them.

Customer reporting is achieved through a GET call on /report/customer/(id) where the id represents the id of the customer requesting the report. The REST call uses the optional query parameters of start and end dates to allow the customer to get reports in certain date ranges. A list of transactionDTOs will be returned on a successful call, which enables the user application to show the data in the desired format for the user. In case of a bad date, an HTTP 400 Bad Request will be returned to the user explaining why the date format was wrong.

Merchant reporting is achieved in the same way as the customer reporting, but with the URL /report/merchant/(id), where id is the merchant id in DTUPay. The same optional start and end date queries are used, but the list of transactionDTOs will have the identifying information of the customer id and balance removed. In case of a bad date, an HTTP 400 Bad Request will be returned to the user explaining why the date format was wrong.