Project Number 683612

D 3.1 Architecture design for the VIMpay API

1.2

06 October 2015

Final

Public distribution

petaFuel

Project Partner Contact Information

petaFuel GmbH
Ludwig Adam
Muenchnerstrasse 4
85354 Freising
Germany
Tel: +49 8161 40 60 202
E-Mail: ludwig.adam@petafuel.de

Table of Content

Document Control

Version	Status	Date
1.0	First document and outline	25 August 2015
1.1	Document for review	16 September 2015
1.2	Final	06 October 2015

Executive Summary

This document constitutes deliverable *D 3.1 Architecture design for the VIMpay API* of Work Package 3 (WP3) of the VIMpay project.

The purpose of this deliverable is to describe the general (technical) architecture of the API that is used for the communication between the VIMpay app (and in a later stage third parties) and the petaFuel backend systems for VIMpay. It highlights several design decisions and details on implementation specifics.

1 Introduction and scope of this deliverable

1.1 pfREST- the open API provided for VIMpay

In the initial proposal for the VIMpay project and the subsequent Description of Action we have highlighted that the smartphone application will communicate with the petaFuel systems through a dedicated open interface (an API) as highlighted in the following figure:

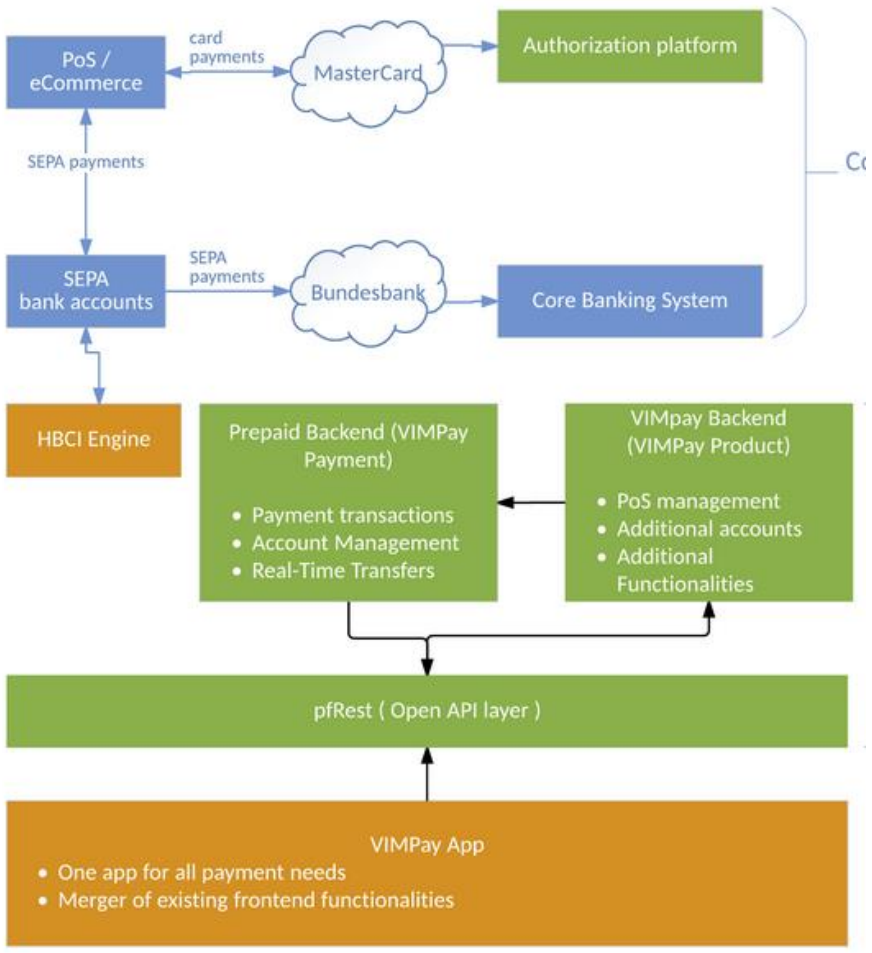


Figure 1 General VIMpay architecture as shown in DoA

In further references, we will call this API layer **pfREST** (for petaFuel REST Interface).

While petaFuel already employs several purpose-built internal or customer-facing interfaces connected to the processing infrastructure, it was decided that a new interface should be developed in scope of the project.
The reason for this decision - apart from technical details - is that we strive to create an interface that directly implements the EU call for open interfaces to banking applications ([1]): It is the goal for the API work package to deliver a connection endpoint not only for the petaFuel-Internal management of VIMpay functionalities but also to open this interface to third parties that wish to connect to VIMpay (this will be so called **VIMpay connected** applications). Examples for such third parties are financial management applications that want to list transactions of a users' VIMpay card or merchants directly using the VIMpay payment options.

Therefore, the API functionalities of VIMpay can be categorized into three different "usage layers" (formerly referenced to as namespaces):

- The **public layer** for API calls that can be done from any person and client.
  These calls may not return sensitive information.
- The **private or authenticated** layer for calls that are user-specific and done by an authenticated user
- The **third party** layer for calls that are connected to a user but not done by the user herself but by a third-party

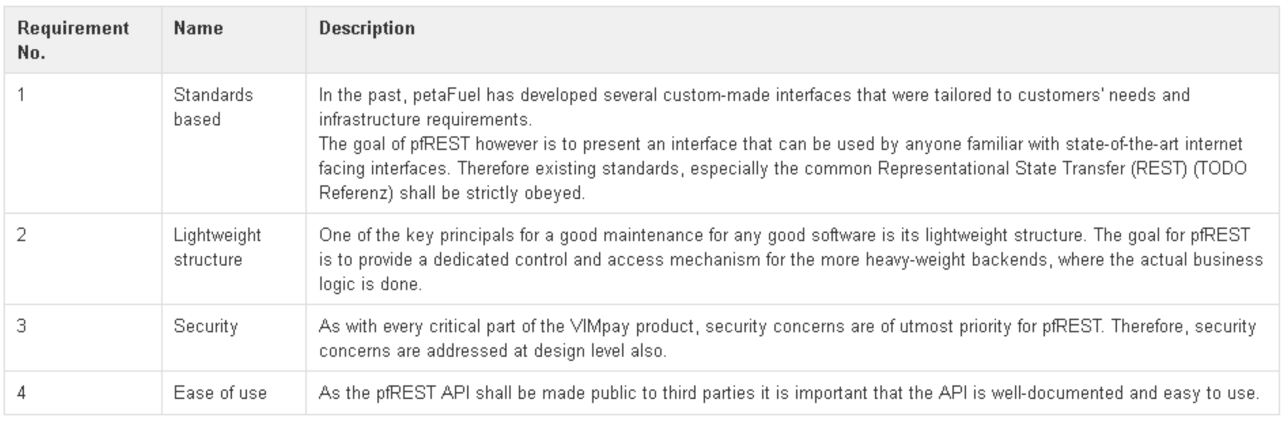
1.2 Scope of this deliverable

Within this deliverable the overall architecture for the pfREST API will be presented along with a description of fundamental implementations necessary to reflect this concept.
In particular we will highlight the access concept and the security concept within pfREST along with some examples on the usage.

2 Architecture design for VIMpay API

2.1 Overall design requirements

While the functional requirements of the VIMpay API are defined by the requirements to the VIMpay and Processing Backend (i.e. the backend functions are provided through calls to pfREST), there are several design requirements that influence the architecture of pfREST:



Requirement No.	Name	Description
1	Standards based	In the past, petaFuel has developed several custom-made interfaces that were tailored to customers' needs and infrastructure requirements. The goal of pfREST however is to present an interface that can be used by anyone familiar with state-of-the-art internet facing interfaces. Therefore existing standards, especially the common Representational State Transfer (REST) (TODO Referenz) shall be strictly obeyed.
2	Lightweight structure	One of the key principals for a good maintenance for any good software is its lightweight structure. The goal for pfREST is to provide a dedicated control and access mechanism for the more heavy-weight backends, where the actual business logic is done.
3	Security	As with every critical part of the VIMpay product, security concerns are of utmost priority for pfREST. Therefore, security concerns are addressed at design level also.
4	Ease of use	As the pfREST API shall be made public to third parties it is important that the API is well-documented and easy to use.

The implementation of the pfREST API follows these design requirements in the form of principles and design guidelines.

2.2 Environment

petaFuels processing platform and infrastructure is built as an Enterprise Java system.
This means in general that programs are implemented in Java using Java EE 6 framework services and run in an Enterprise Java certified application server environment.

Following that the VIMpay REST API is implemented in Java EE and the Java API for RESTful Web Services (JAX-RS, [2]) is used.
The application runs inside a JBoss application server with Java EE support.

The following figure shows the communication flow between the VIMpay application and the servers as it is implemented in the production environment:

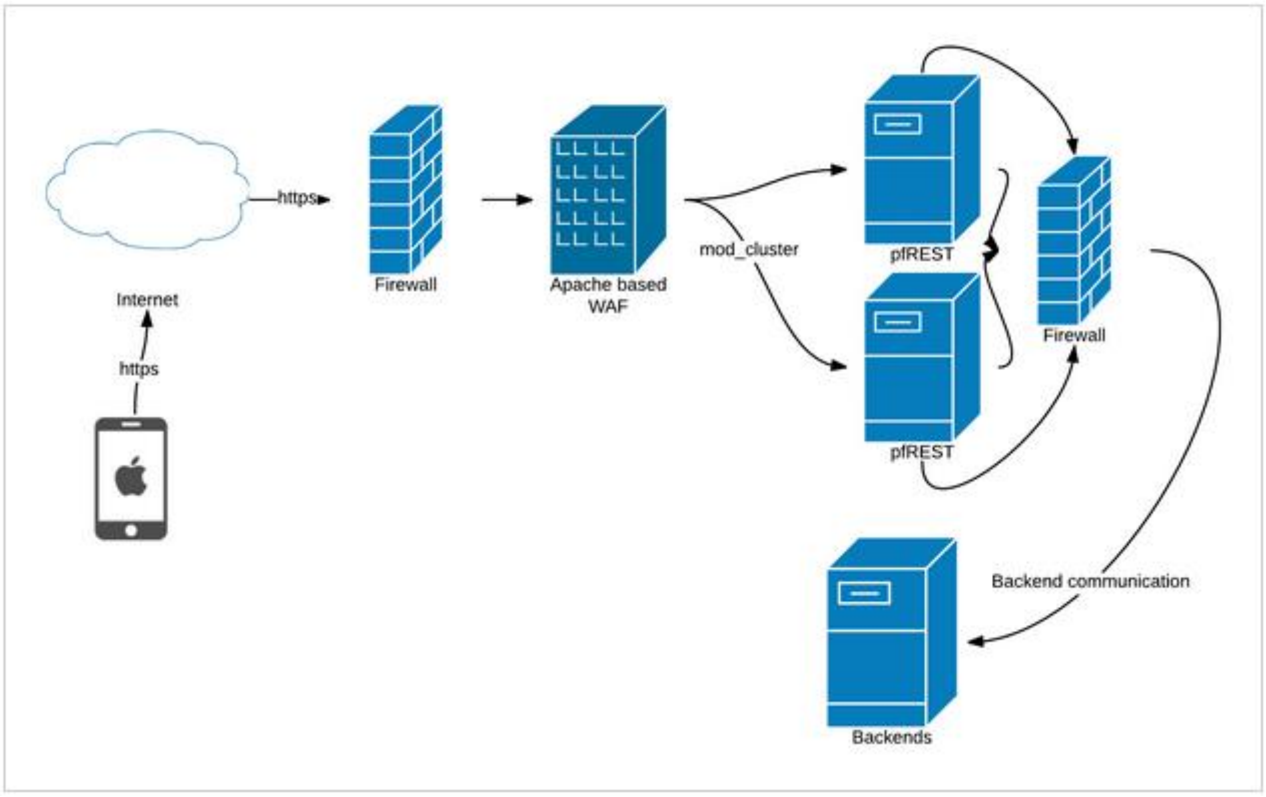


Figure 2 Communication flow for pfREST calls

Currently the setup is as following:

- The pfREST JAX-RS compliant application is deployed on two frontend JBoss Application servers which are clustered to provide failover mechanisms
- The frontend servers / the API communicates with a backend in an internal network
- Communication between the App and the API is done solely through encrypted HTTPS communication and tunneled through a packet filtering firewall and a dedicated Web Application Firewall (WAF)

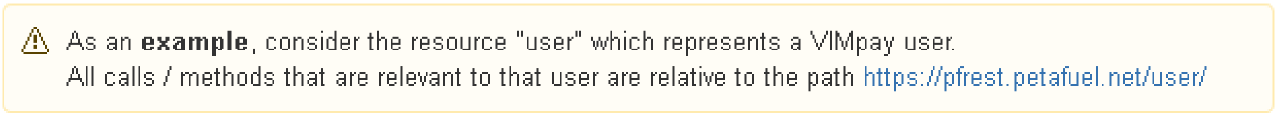
This setup provides a secure setup, both from data security as well as fault tolerance perspective.

3 Structure & Implementation

3.1 Namespaces and resource addressing by URIs

In a RESTful API method calls are done by accessing Uniform Resource Identifiers (URIs) via the HTTP protocol.
While URIs are very similar to URLs of common webpages (and are represented by the same strings) they not necessarily point to webpages but rather to Internet resources, in this case API methods.

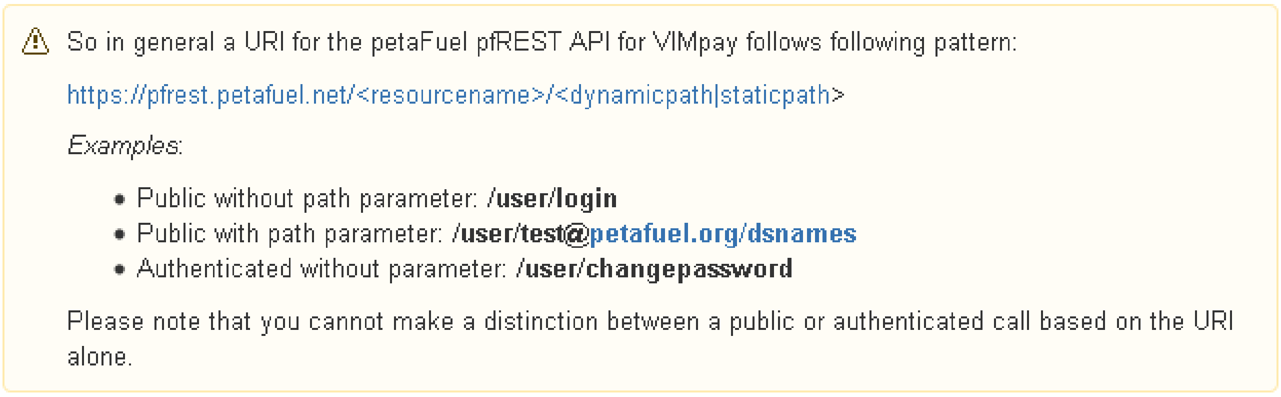
Therefore, the API has a root URI, https://pfrest.petafuel.net and all methods / resources are addressed by URIs relative to this URI.
There are no consistent specifications or best practices on how to structure the URI calls as it is dependent on the nature of the application but in general we are following the pattern that resources are first class entities and actions are relative to the resource.


⚠ As an **example**, consider the resource "user" which represents a VIMpay user.
All calls / methods that are relevant to that user are relative to the path https://pfrest.petafuel.net/user/

Please note: In the initial DoA we have defined the different access levels such as public, authenticated and third-party by different namespaces (e.g. https://pfrest.petafuel.net/public/user for public methods and https://pfrest.petafuel.net/private/user for private actions on user resource objects).
This has been revised as our implementation of the security concept (see Auth Level discussion below) does not need the distinction in the namespace and it would break the initial resource-oriented design.
Nevertheless we still make a logical distinction between these different access levels.

3.2 Path parameters

Underneath the root URL of the REST API it is only allowed to use resources like *user* or *instantreplenishment*.
Each resource has its own namespace and provides specific functionality via additional paths.
The additional path may consist of *static* or *dynamic parts.* Dynamic parts are used as parameters, so called path parameters.


⚠ So in general a URI for the petaFuel pfREST API for VIMpay follows following pattern:

https://pfrest.petafuel.net/<resourcename>/<dynamicpath|staticpath>

Examples:

- Public without path parameter: **/user/login**
- Public with path parameter: **/user/test@petafuel.org/dsnames**
- Authenticated without parameter: **/user/changepassword**

Please note that you cannot make a distinction between a public or authenticated call based on the URI alone.

3.3 Implementation of an RESTful interface

The implementation of the interface is done using the JAX-RS framework ([2]) .
In particular we implement one Java class per Resource and implement the available methods as corresponding methods in that java class.

3.3.1 Calls & Responses in line of the HTTP protocol

While REST is no official standard per se there are de-facto standard conventions any RESTful interface should follow.
In particular these are:

- Communication is done using HTTP
- Methods do matter: There are different meanings for different HTTP Methods when calling a resource.
  For example a PUT method on a resource should create a corresponding resource while a HTTP GET should retrieve a resource
- Error messages and server feedback should follow the HTTP status codes

These conventions are being followed in the pfREST architecture.
With regards to the different method calls, pfREST uses the following conventions:

Method	Meaning
GET	Retrieves a resource or a value of a resource or calls a side-effect free method on the resource
POST	Adds properties to resource, uploads data, calls methods with side-effects on a resource
OPTIONS	Retrieves a list of possible calls on a resource.

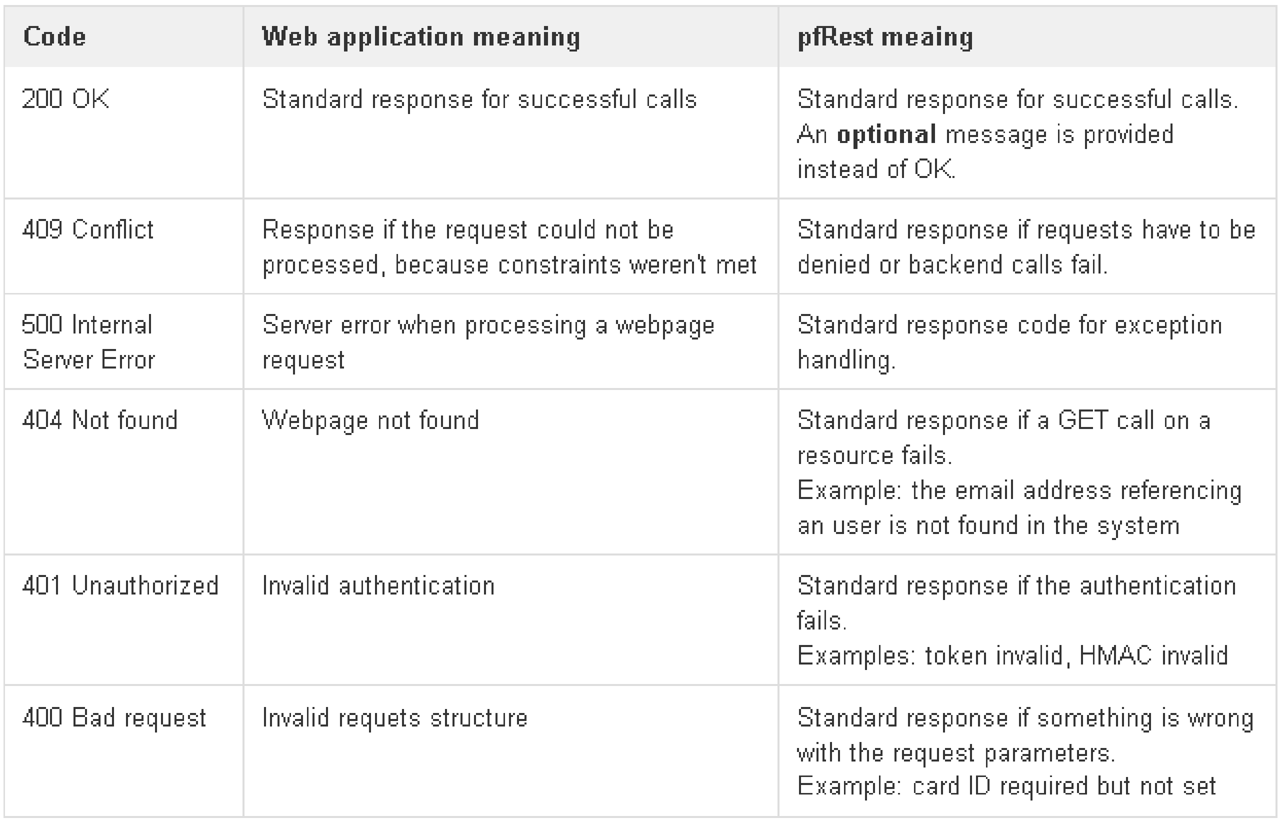
Table 1 HTTP Methods supported by pfREST

Please note that of the current state we have decided not to support PUT or DELETE method calls on resources.
The reasoning behind that is following:

- First of all, in historic context of standard web applications PUT and DELETE operations were only used for special applications such as WebDAV.
  Therefore, most web application firewalls block PUT and DELETE operations by default- we would need to define explicit exceptions for specific resources.
- Second, looking at the architecture of VIMpay, resources are not directly created or deleted based on the method call to the API. In general, the creation or deletion is only triggered and the actual operation (such as creating a user) is done asynchronously for the user.
  For examples IDs - which are necessary for PUT requests- are controlled by our backend and not by the client calling the interface. Therefore we have decided to use POST and GET operations instead.

Each method call produces different HTTP **response codes** based on the result of the call.
While the standard HTTP 200/OK response code for successful execution is common for all resource calls, error codes are of course method specific.

Currently in use are:



Code	Web application meaning	pfRest meaing
200 OK	Standard response for successful calls	Standard response for successful calls. An **optional** message is provided instead of OK.
409 Conflict	Response if the request could not be processed, because constraints weren't met	Standard response if requests have to be denied or backend calls fail.
500 Internal Server Error	Server error when processing a webpage request	Standard response code for exception handling.
404 Not found	Webpage not found	Standard response if a GET call on a resource fails. Example: the email address referencing an user is not found in the system
401 Unauthorized	Invalid authentication	Standard response if the authentication fails. Examples: token invalid, HMAC invalid
400 Bad request	Invalid requets structure	Standard response if something is wrong with the request parameters. Example: card ID required but not set

Table 2 Current response codes of pfREST

3.3.2 Standard response format

pfREST supports different output formats in general but the standard format is JSON ([3]) .

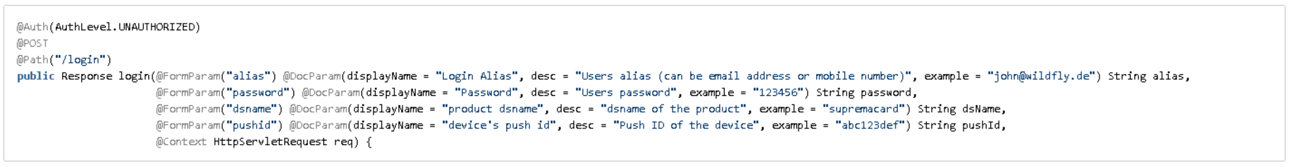
3.4 Documentation

We strongly believe that a good API must have an intuitive and good documentation- only then it can be used effectivly by other developers. With JavaDoc Java already supports a very good streamlined documentation mechanism that allows developers to directly annotate methods with documentation, which is then compiled into a comprehensive code documentation. Following that approach we have implemented the functionality that the documentation for a resource and its method call is directly available via REST calls:

- Once a function is implemented each method parameter is annotated using the self-definied @DocParam Annotation (comparable to the @JavaDoc annotation)
- A developer / user of the API may call any resource with the OPTIONS method call and retrieve the most current documentation for this method.

Example:

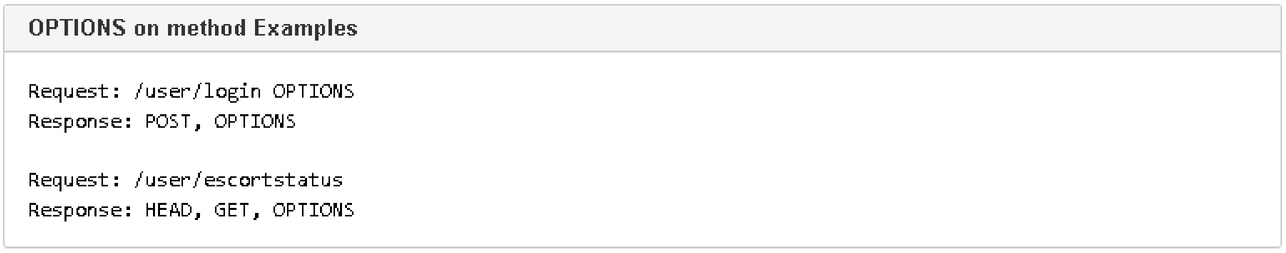
Documentation for user/login:



```
@Auth(AuthLevel.UNAUTHORIZED)
@POST
@Path("/login")
public Response login(@FormParam("alias") @DocParam(displayName = "Login Alias", desc = "Users alias (can be email address or mobile number)", example = "john@wildfly.de") String alias,
                      @FormParam("password") @DocParam(displayName = "Password", desc = "Users password", example = "123456") String password,
                      @FormParam("dsname") @DocParam(displayName = "product dsname", desc = "dsname of the product", example = "supremacard") String dsName,
                      @FormParam("pushid") @DocParam(displayName = "device's push id", desc = "Push ID of the device", example = "abc123def") String pushId,
                      @Context HttpServletRequest req) {
```

Figure 3 Code snippet showing in line documentation

When now calling a specific user resource method with OPTIONS the pfREST API returns the supported HTTP methods, e.g. POST or GET:



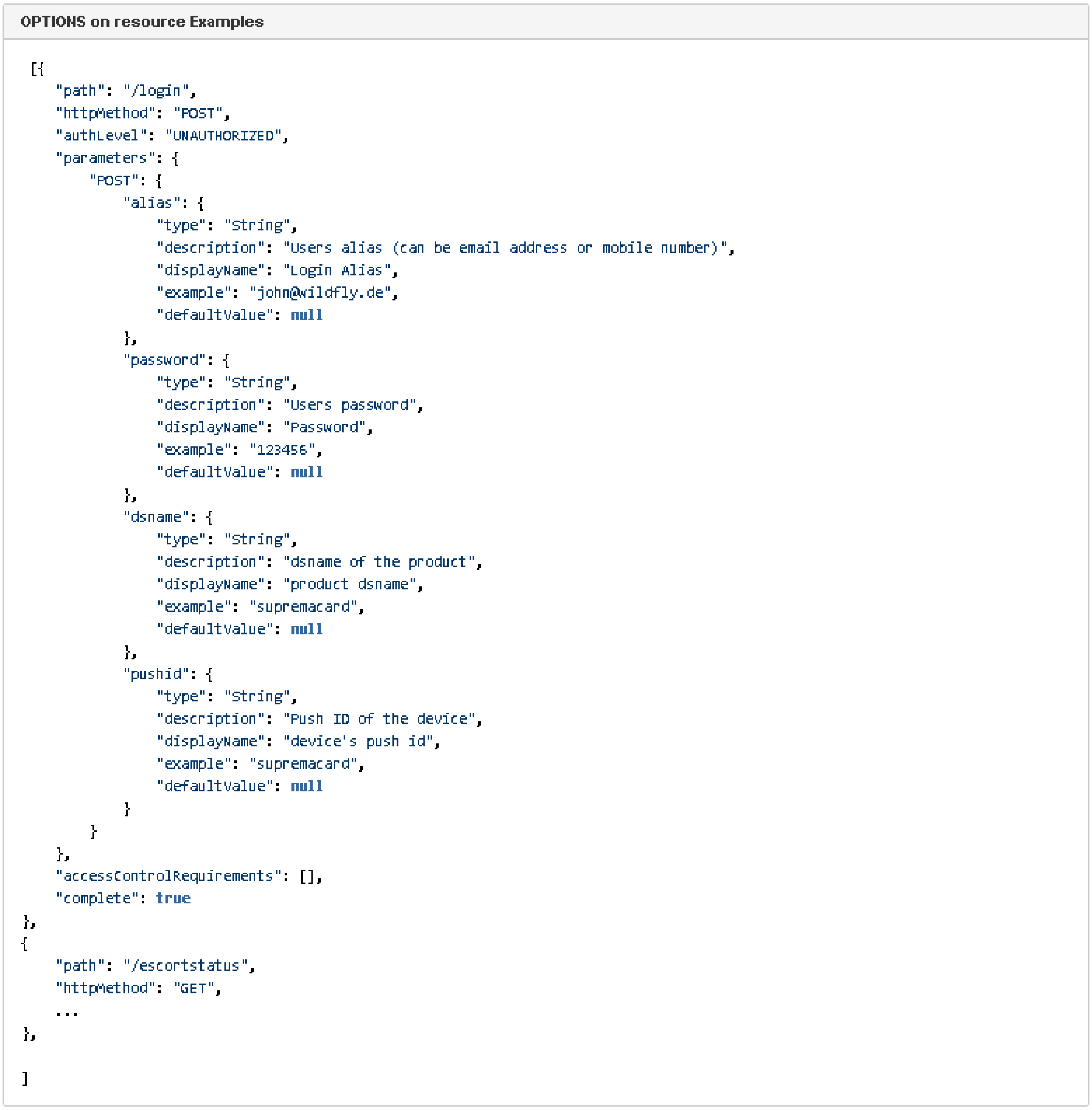
OPTIONS on method Examples

```
Request: /user/login OPTIONS
Response: POST, OPTIONS

Request: /user/escortstatus
Response: HEAD, GET, OPTIONS
```

Figure 4 OPTIONS call on pfREST

When calling the user resource directly with OPTIONS the pfREST API returns a description for every method implemented in this resource.



OPTIONS on resource Examples

```
[{
    "path": "/login",
    "httpMethod": "POST",
    "authLevel": "UNAUTHORIZED",
    "parameters": {
        "POST": {
            "alias": {
                "type": "String",
                "description": "Users alias (can be email address or mobile number)",
                "displayName": "Login Alias",
                "example": "john@wildfly.de",
                "defaultValue": null
            },
            "password": {
                "type": "String",
                "description": "Users password",
                "displayName": "Password",
                "example": "123456",
                "defaultValue": null
            },
            "dsname": {
                "type": "String",
                "description": "dsname of the product",
                "displayName": "product dsname",
                "example": "supremacard",
                "defaultValue": null
            },
            "pushid": {
                "type": "String",
                "description": "Push ID of the device",
                "displayName": "device's push id",
                "example": "supremacard",
                "defaultValue": null
            }
        }
    },
    "accessControlRequirements": [],
    "complete": true
},
{
    "path": "/escortstatus",
    "httpMethod": "GET",
    ...
},

]
```

Figure 5 Documentation output returned by resource call

4 Security

As with any VIMpay component, security is a first-level concern when designing and implementing the API. We make a distinction between **infrastructure based security** and **application level security**. While on the first we are mainly concerned with ensuring that the servers and communication is not compromised, for the latter we are concerned with data and user level security such as:

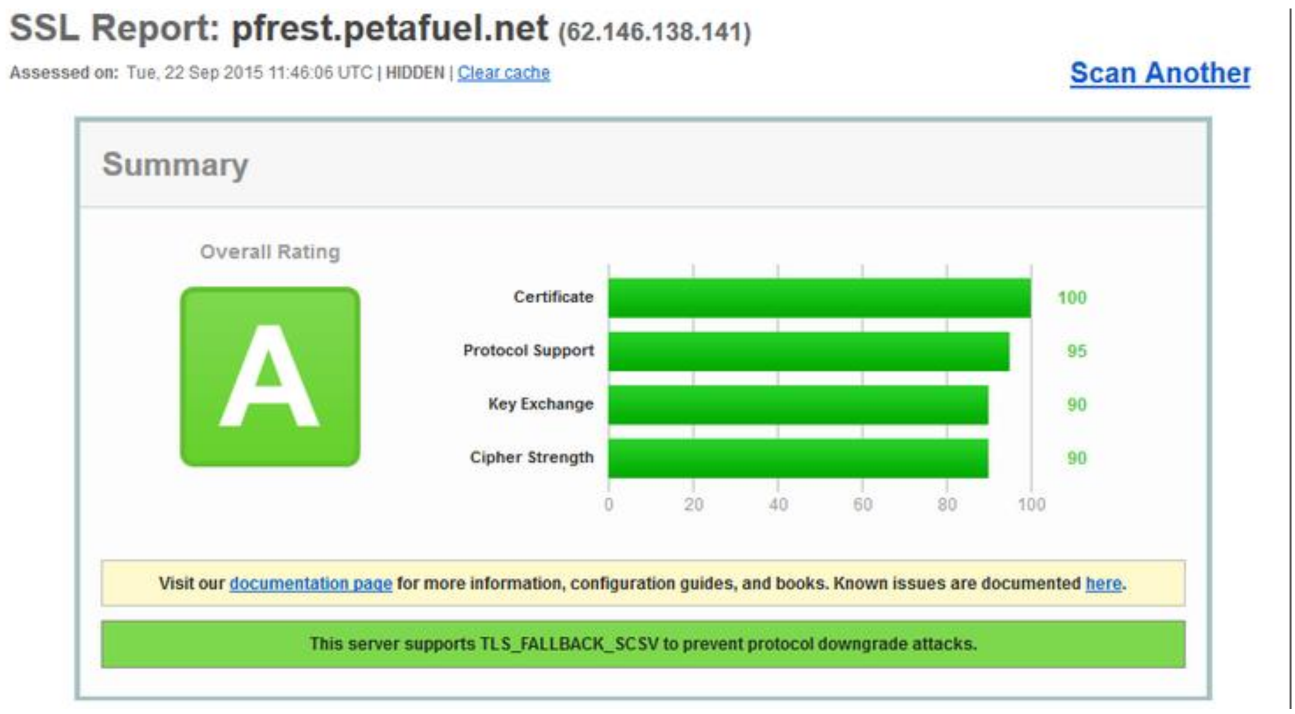
- User authentication: Ensuring that users are properly authenticated
- Data authenticity: Ensuring that data has not been manipulated
- Access control: Ensuring, that requests can only be made for the right data (i.e. the users' own data and no foreign data)

4.1 Security in Infrastructure

In general the pfREST API is treated as any web application that is available in the production environment. Therefore, all the security requirements imposed by the high PCI-DSS and ISO 27001 standards ([4]) also apply to the pfREST API.

- Calls can only be made through https
- The SSL configuration must follow the latest standards
- All requests are filtered through a web application firewall
- The infrastructure is pentested on a regular (quarterly basis)

The following figure shows the current A-Level SSL configuration of the API endpoint.


SSL Report: pfrest.petafuel.net (62.146.138.141)

Assessed on: Tue, 22 Sep 2015 11:46:06 UTC | HIDDEN | Clear cache

Visit our documentation page for more information, configuration guides, and books. Known issues are documented here.

This server supports TLS_FALLBACK_SCSV to prevent protocol downgrade attacks.

Figure 6 SSL configuration of pfREST endpoint

4.2 Overall description of authentication and access processes

When it comes to authentication and access control in the context of VIMpay following aspects have to be considered:

- Based on the design requirements there is a distinction to be made between public, authenticated (private) and third party access
- Multiple devices calling the API for the same user account shall be supported as required by the business requirements outlined in D 5.1 ([5])
- Some method calls are specific to a given card / user account while other methods are more generic
- A standard user should have access only to his data not the data of other users. Therefore calls to another users' resources must not be possible
- Credentials for calling the API should be cache-able as required by the business requirements ([5])

While there are standard mechanisms for authentication and access controls available (such as OAuth), they do not cater for the specifics of VIMpay. It has therefore been decided to introduce a proprietary security level to cater for all the needs of VIMpay.

Our token-based-approach can be described in general as follows:

- We defined an **Auth Level** which describe the level of security / privacy for each method call. Currently we have three Auth Levels
  - AuthLevel.Authorized corresponds to private access
  - AuthLevel.Authorized_HMAC corresponds to private access and message authentication
  - AuthLevel.Unauthorized corresponds to public access.

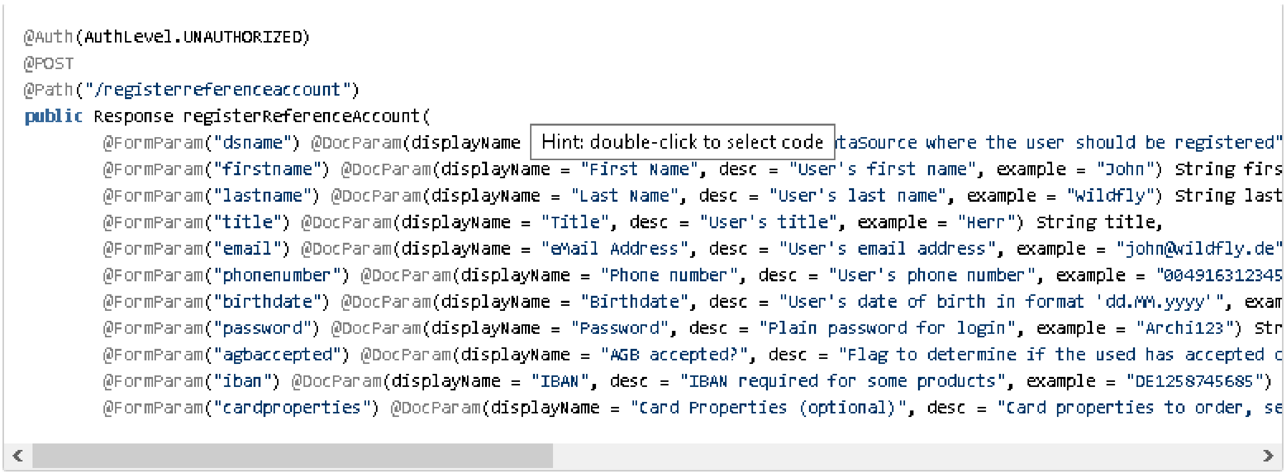
  On method level it is defined and checked, which access level is required for executing the method.
- While calls to AuthLevel.Public calls can be done by any user, Authenticated calls require an authentication of the calling user
- An user authenticates himself by transmitting a previously acquired session-token for each call- this token is then checked prior to the execution of the method.
- Linked to this session-token is a access control list that defines, which access the authenticated caller of the method has.

In the following sections we detail the implementation of this concept.

4.3 Details in user authentication

4.3.1. Public and private namespace: Auth Level

The authentication level of each method is defined by the developer by the @Auth annotation, which is then checked before the method call is executed.



```
@Auth(AuthLevel.UNAUTHORIZED)
@POST
@Path("/registerreferenceaccount")
public Response registerReferenceAccount(
        @FormParam("dsname") @DocParam(displayName  Hint: double-click to select code  taSource where the user should be registered"
        @FormParam("firstname") @DocParam(displayName = "First Name", desc = "User's first name", example = "John") String firs
        @FormParam("lastname") @DocParam(displayName = "Last Name", desc = "User's last name", example = "Wildfly") String last
        @FormParam("title") @DocParam(displayName = "Title", desc = "User's title", example = "Herr") String title,
        @FormParam("email") @DocParam(displayName = "eMail Address", desc = "User's email address", example = "john@wildfly.de"
        @FormParam("phonenumber") @DocParam(displayName = "Phone number", desc = "User's phone number", example = "004916312345
        @FormParam("birthdate") @DocParam(displayName = "Birthdate", desc = "User's date of birth in format 'dd.MM.yyyy'", exam
        @FormParam("password") @DocParam(displayName = "Password", desc = "Plain password for login", example = "Archi123") Str
        @FormParam("agbaccepted") @DocParam(displayName = "AGB accepted?", desc = "Flag to determine if the used has accepted c
        @FormParam("iban") @DocParam(displayName = "IBAN", desc = "IBAN required for some products", example = "DE1258745685")
        @FormParam("cardproperties") @DocParam(displayName = "Card Properties (optional)", desc = "Card properties to order, se
```

Figure 7 Example of the @Auth annotation

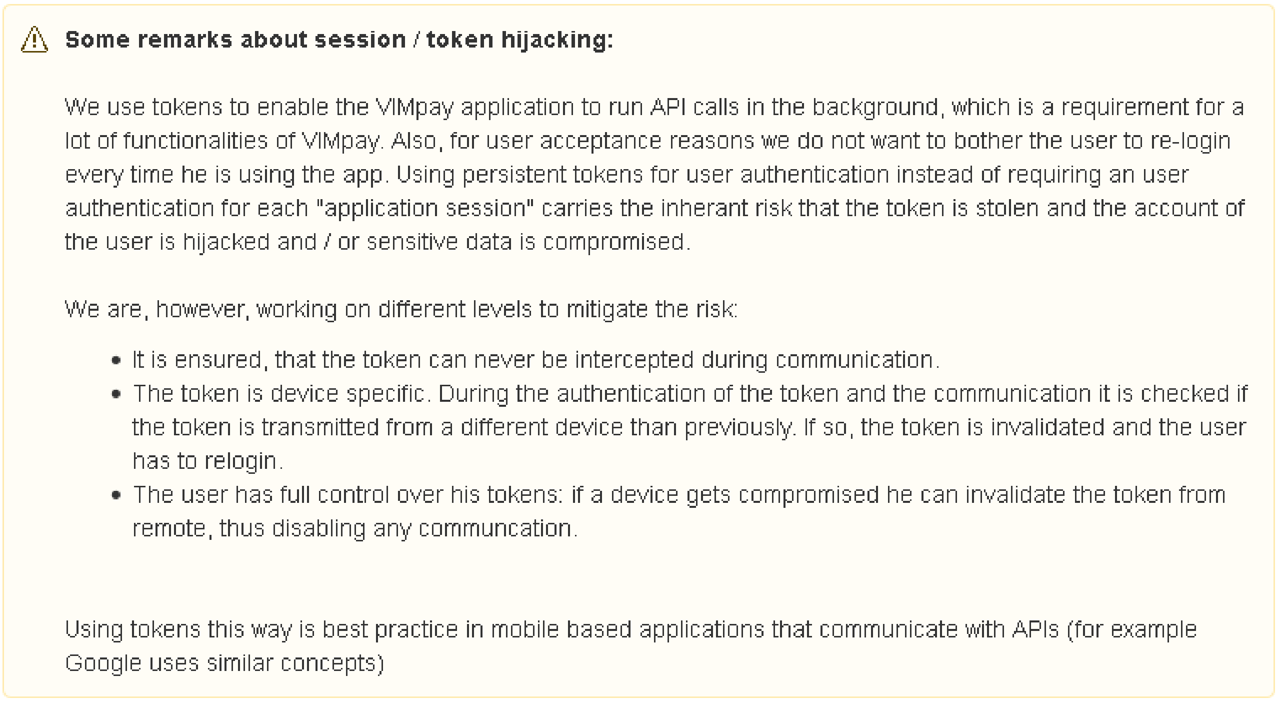
Please note: as the authentication level is defined on a method level, a specific resource may contain both public and private methods of course.

4.3.2 Authentication through Session Tokens: Concept & Login

Whether a caller of an API method is allowed to call a method is determined by two aspects: First by the AuthLevel of the method in question and second by the fact if the caller is providing a **valid session token** along its request.
A session token is basically a unique String that is transmitted by the REST client in form of a HTTP-header. It is centrally checked and can be acquired by:

- Using a user specific login-call providing login data. (e.g. /user/login with email and password)
  This automatically generates a session token that is linked to the users' card account and allows for card specific access only
- By a static entry generated by other management interfaces or manual entry.

Although we call these tokens session-tokens they are not session-specific as in the common understanding of Session IDs in web applications.
Tokens can and shall be re-used - they only expire after some time. Tokens can be stored locally on the device and used for further calls.



Some remarks about session / token hijacking:

We use tokens to enable the VIMpay application to run API calls in the background, which is a requirement for a lot of functionalities of VIMpay. Also, for user acceptance reasons we do not want to bother the user to re-login every time he is using the app. Using persistent tokens for user authentication instead of requiring an user authentication for each "application session" carries the inherant risk that the token is stolen and the account of the user is hijacked and / or sensitive data is compromised.

We are, however, working on different levels to mitigate the risk:

- It is ensured, that the token can never be intercepted during communication.
- The token is device specific. During the authentication of the token and the communication it is checked if the token is transmitted from a different device than previously. If so, the token is invalidated and the user has to relogin.
- The user has full control over his tokens: if a device gets compromised he can invalidate the token from remote, thus disabling any communcation.

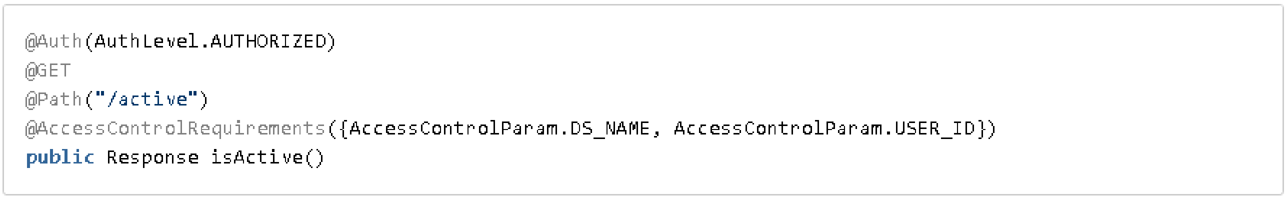
Using tokens this way is best practice in mobile based applications that communicate with APIs (for example Google uses similar concepts)

4.4 Access control and access control requirements

In the context of VIMpay most operations are performed on **card accounts and / or user accounts**. As the Prepaid Backend can also be used not only for the VIMpay product but for other PayCenter cards there is also a product descriptor necessary (although it will be "vimpay" for all calls related to VIMpay).
So in general, for most method calls it must be known which **card account and / or user account is affected**. However, some functions (such as the registration) do not require the specification of card or user account.

As we already have the authentication mechanism in place we have solved this problem in following way:

- For each method call it is specified, if the user account and / or card account needs to be available. This is done by the so called AccessControlRequirements- Annotation:



```
@Auth(AuthLevel.AUTHORIZED)
@GET
@Path("/active")
@AccessControlRequirements({AccessControlParam.DS_NAME, AccessControlParam.USER_ID})
public Response isActive()
```

Figure 8 Example of the AuthControlRequirements-Annotation

- We link to each token a access control list that lists the card IDs and user IDs (and product names) the caller with that token has access to.
  An user may have access to one specific card ID and user ID only or this access control list may list several card IDs or even a wildcard, enabling full access
- Any method that defines an access control requirement needs that specific parameter **to be uniquely defined**.
  - In case of one specific card ID and user ID linked to the session token, this is already defined

  - However, if the access control lists several possible card IDs, the card ID must be specified as part of a dynamic parameter.

4.5 Summary of the authentication and access control access concept

To summarize these concepts the following table shows the different possible constellations for different calls:

Auth level of method called	Access control requirement	Session token provided	Access Control list entry	Path	Result	Remark
Unauthorized	None	N/A	N/A	example/public	OK	Standard public call
Authorized	None	None	None	example/privatecall	NOT OK	Not authenticated
Authorized	None	Yes	cardID|userID|productname	example/privatecall	OK	
Authorized	Card ID	Yes	cardID|userID|productname	example/privatecall	OK	Card ID is provided directly through access control list
Authorized	Card ID	Yes	cardID|userID|productname	example/privatecall /cardid2	NOT OK	User provides card ID he has no access to
Authorized	Card ID	Yes	* | * | productname	example/privatecall/	NOT OK	Card ID not uniquely provided
Authorized	Card ID	Yes	* | * | productname	example/privatecall /cardid3	OK	User has access to all card ids and provides a card in the call

Table 3 Different configurations for SessionToken and AccessControlRequirements

4.6 Message authentication (MAC)

For certain cases we would like to make sure that the calls have not been done manually by the user and / or the data integrity of the parameters has not been compromised by an error.
We have therefore introduced another authentication level that requires an HMAC to be submitted along with the call request. The HMAC is a SHA-256 hash of the message content using the session token as a key. The message / call is rejected if the Hash value does not correspond to the values / parameters submitted in the call.

4.7 Central implementation

Previously we have mentioned a central check for the authentication and access concept. This is done by a central filter component called SecurityRequestFilter which is called by default for every request made by the API.
The checks done by this filter correspond to the checks for the different authentication and access concepts hightlighted above:

- The Auth Level of the call is checked
- If Authentication is required, the session token is checked for presence and validity
- If HMAC is required, the HMAC is checked for presence and validity
- The AccessControl assigned to the token is loaded
- The AccessControlRequirements of the method are checked
    - Against uniqueness of provided card id and user id (see above)
    - Against validity of the provided cardid in the context of the access control list

5 References

[1] European Central Bank, „Recommendations for "Payment Account Access" Services,“ European Central Bank, Brussels, 2013.

[2] O. inc., „JSR 311: JAX-RS: The JavaTM API for RESTful Web Services,“ 2009. [Online]. Available: https://jcp.org/en/jsr/detail?id=311.

[3] Crockford, „RFC 4627 JSON,“ 2009. [Online]. Available: https://tools.ietf.org/html/rfc4627.

[4] incits , *INCITS/ISO/IEC 27001-2005.*

[5] p. GmbH, „Business requirements for version 1 of the VIMpay app,“ 2015.