**Project – Banking System**

Edureka!

**Java, Java EE & SOA Certification Training**

Student: AmanVerma

Date: 29-10-2022

**Table of Contents**

1. [Learning Objectives, Aim, Purpose, Abstract 3](#_bookmark0)
2. [Architecture and components 4](#_bookmark1)
3. [MVC Controller , Model, Views 5](#_bookmark2)
4. [Web Application: Functionality and User interface 11](#_bookmark3)
   1. [User logging screen 11](#_bookmark4)
   2. [Main Menu screen 12](#_bookmark5)
   3. [Account Creation 13](#_bookmark6)
   4. [Debit – Credit 16](#_bookmark7)
   5. [Display Statement 20](#_bookmark8)
   6. [Credit Card Transaction 21](#_bookmark9)
5. [Database 27](#_bookmark10)
   1. [JDBC (Java database connectivity) 27](#_bookmark11)
   2. [DataBase Design, Tables 27](#_bookmark12)
6. [Bibliography 32](#_bookmark13)

# Learning Objectives, Aim, Purpose, Abstract

Learn and master the basic and advanced concepts of core Java SE & Java EE along with popular frameworks like Hibernate, Spring and SOA. Gain expertise in the concepts like Java Array, Java OOPs, Java Function, Java Loops, Java Collections, Java Thread, Java Servlet, Java Design Patterns, and Web Services using industry use-cases

To apply these new skills and concepts learned, a banking project need to be developed.

As ***requirements definition*** for the project , the following modules should be implemented :

Module 1 : This module accepts user id and password and authenticates the given credentials with the database using hibernate.

Module 2 : By accepting the type of account (SB / Current A/c) and user's details, account will be created.

Module 3 : Perform the debit and credit transactions.

Module 4 : Accept credit card information with the desired details and authorize the credit card amount using web services.

Module 5 : Display transactions of the account based on given date range (From a specific date to a specific date).

*Web Services and Project*

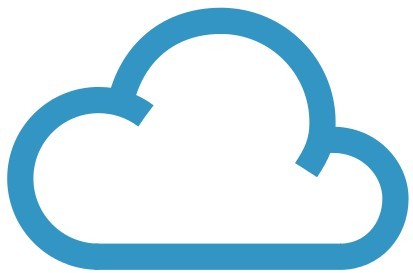
*Learning Objectives - In this module, you will learn SOA and implementation of web services. We will also discuss how to develop a project using Spring and Hibernate. This is a banking project with web services.*

# Architecture and components

* + Spring MVC Model - View - Controller
    - Front Side (JSP, HTML, JavaScript, JSTL)
    - Back-Side JEE Spring MVC
  + SOA Architecture witth a RESTful Web Service
  + MariaDB (MySQL) Data - Base
  + JDBC (Transaction and batch processing)
  + Server: TomCat 8.5x

**IP3:PortNumber3**

**IP2:PortNumber2 (127.0.0.1:3306)**



RESTful Web Server

**IP4:Port4**

Data Base B

Credit Card Server

JEE

credit\_card\_service

MySQL

**IP1:PortNumber1**

**(127.0.0.1:8080)**

**Internet**

Server TomCat

JEE SPRING MVC

Data Base A Server

ProjectJEE

MySQL

User 2 Computer

Front Side HTML, JavaScript

User 1 Computer

Front Side HTML, JavaScript

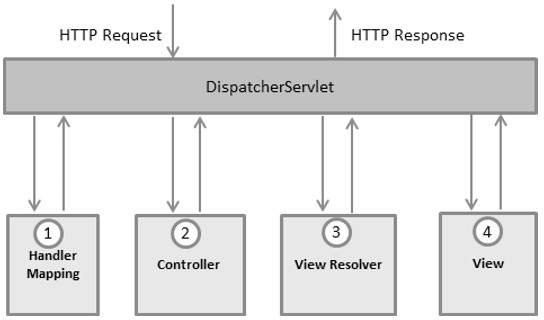
# MVC Controller , Model, Views

The Spring Web MVC framework provides Model-View-Controller (MVC) architecture and ready components that can be used to develop flexible and loosely coupled web applications. The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* + The ***Model*** encapsulates the application data and in general they will consist of POJO.
  + The ***View*** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
  + The ***Controller*** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

### The DispatcherServlet

The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that handles all the HTTP requests and responses. The request processing workflow of the Spring Web MVC DispatcherServlet is illustrated in the following diagram.



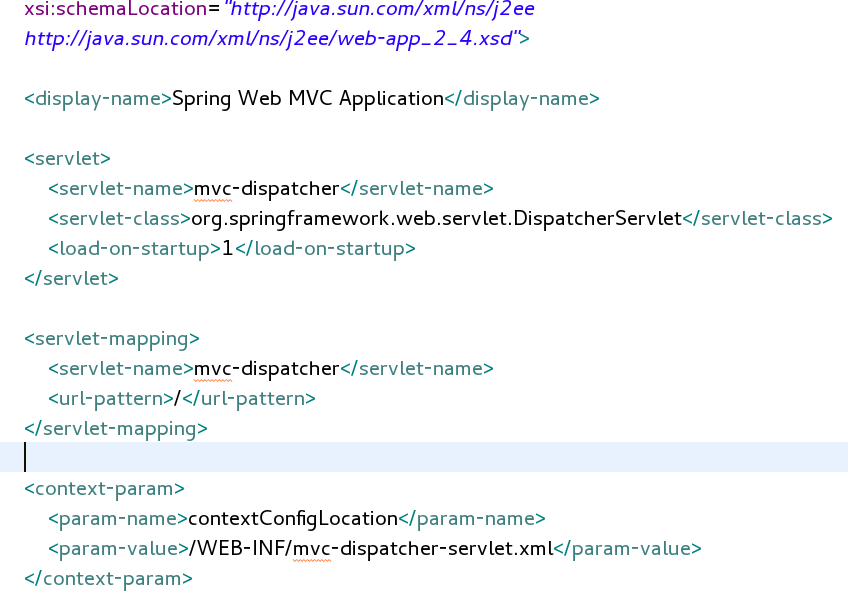
Following is the sequence of events corresponding to an incoming HTTP request to DispatcherServlet

* + After receiving an HTTP request, DispatcherServlet consults the HandlerMapping to call the appropriate Controller.
  + The Controller takes the request and calls the appropriate service methods based on used GET or POST method. The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
  + The DispatcherServlet will take help from ViewResolver to pickup the defined view for the request.
  + Once view is finalized, The DispatcherServlet passes the model data to the view which is finally rendered on the browser.

All the above-mentioned components, i.e. HandlerMapping, Controller, and ViewResolver are parts of WebApplicationContext which is an extension of the plainApplicationContext with some extra features necessary for web applications.

### Required Configuration

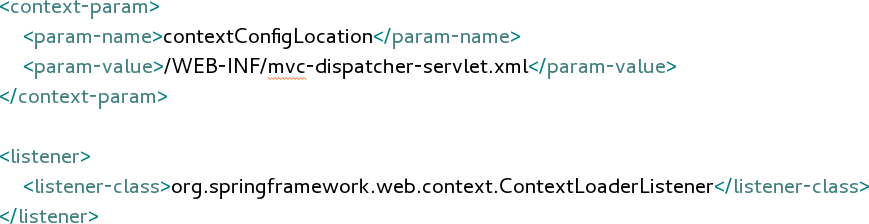
I need to map requests that I want the DispatcherServlet to handle, by using a URL mapping in the web.xml file. The following is an example to show declaration and mapping for mvc-dispatcher DispatcherServlet example



The web.xml file is kept in the WebContent/WEB-INF directory of your web application. Upon initialization of mvc-dispatcher DispatcherServlet, the framework will try to load the application context from a file named [servlet-name]-servlet.xml located in the application's WebContent/WEB- INFdirectory. In this case, our file will be mvc-dispatcher-Webservlet.xml.

Next, <servlet-mapping> tag indicates what URLs will be handled by which DispatcherServlet. Here all the HTTP requests are handled by the mvc-dispatcher DispatcherServlet.

We can customize the the application context file name and location by adding the servlet listener ContextLoaderListener in your web.xml file as follows.



Now, let us check the required configuration for mvc-dispatcher-servlet.xml file, placed in the web application's WebContent/WEB-INF directory.

beansxmlns=*"htp:/[www.springframework.org/schema/beans](http://www.springframework.org/schema/beans)"* xmlns:context=*"htp:/[www.springframework.org/schema/context](http://www.springframework.org/schema/context)"* xmlns:mvc=*"htp:/[www.springframework.org/schema/mvc](http://www.springframework.org/schema/mvc)"*

<

xmlns:xsi=*"htp:/[www.w3.org/20](http://www.w3.org/20) 1/XMLSchema-instance"* xsi:schemaLocation=*"*

*htp:/[www.springframework.org/schema/beans](http://www.springframework.org/schema/beans) htp:/[www.springframework.org/schema/beans/spring-beans-3.0.xsd](http://www.springframework.org/schema/beans/spring-beans-3.0.xsd) htp:/[www.springframework.org/schema/context](http://www.springframework.org/schema/context) htp:/[www.springframework.org/schema/context/spring-context-3.0.xsd](http://www.springframework.org/schema/context/spring-context-3.0.xsd) htp:/[www.springframework.org/schema/mvc](http://www.springframework.org/schema/mvc)*

*htp:/[www.springframework.org/schema/mvc/spring-mvc-3.0.xsd](http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd)"*>

<context:component-scanbase-package=*"com.projectJE.common"*/>

<mvc:anotation-driven/>

<beanid=*"viewResolver"*

clas=*"org.springframework.web.servlet.view.InternalResourceViewResolver"*>

<propertyname=*"prefix"*>

<value>/WEB-INF/jsp/</value>

</property>

<propertyname=*"suffix"*>

<value>.jsp</value>

</property>

</bean>

<beanclas=*"org.springframework.context.sup ort.ResourceBundleMesageSource"* id=*"mesageSource"*>

<propertyname=*"basename"*value=*"mesages"*/>

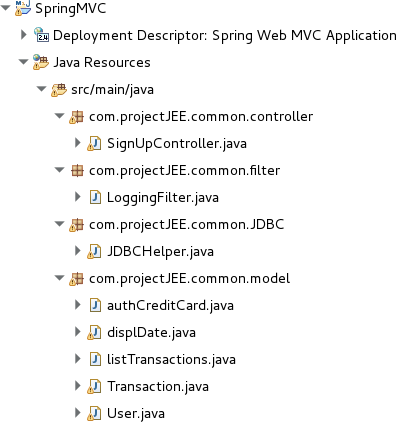
</bean>

/

<beans>

Following are the important points about mvc-dispatcher-servlet.xml file.

* + The [servlet-name]-servlet.xml file will be used to create the beans defined, overriding the definitions of any beans defined with the same name in the global scope.
  + The <context:component-scan...> tag will be use to activate Spring MVC annotation scanning capability which allows to make use of annotations like @Controller and @RequestMapping etc.
  + The InternalResourceViewResolver will have rules defined to resolve the view names.

As per the above defined rule, a logical view named index is delegated to a view implementation located at /WEB-INF/jsp/index.jsp .

\

First a ***Maven project*** with a basic web app template is created. Then the the required libraries such as ***Spring MVC*** (spring-core, spring-web, spring- webmvc, hibernate-validator, servlet- api, javax.websocket-api, jstl, mysql- connector-java) are added in the ***pom.xml***. All dependencies are managed automatically.

***DAO Data Access Object library*** *:* All the data base functions are located in the *com.projectJEE.common.JDBC* package in the ***JDBCHelper.java* class** (open, close connection , CRUD operations…)

All ***model data objects*** exchanged between the views and the controller are placed in the package

*com.projectJEE.common.model*.

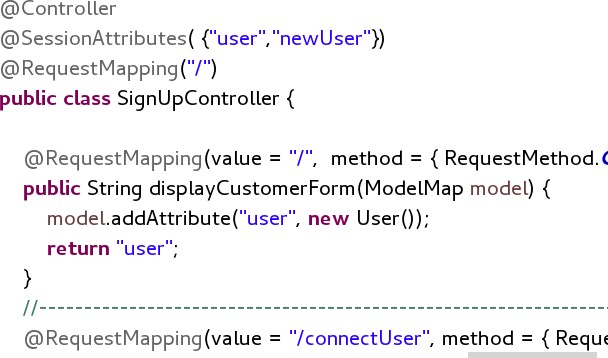
The view resolver is located in the *mvc-dispatcher-servlet.xml*. In this project, all the ***views*** are ***jsp pages*** located under the jsp folder.

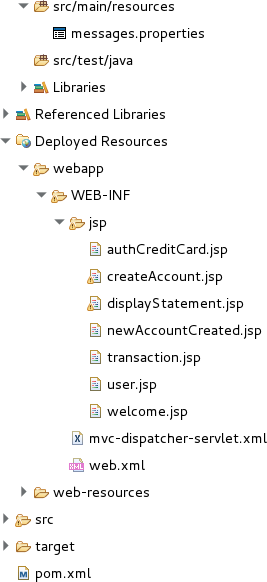
*A* ***filter LoggingFilter.java*** *in the package com.projectJEE.common.filter is used to authenticate the user each request.*



The ***controller*** which will process all the requests is placed in the *SignUpController.java* class in the

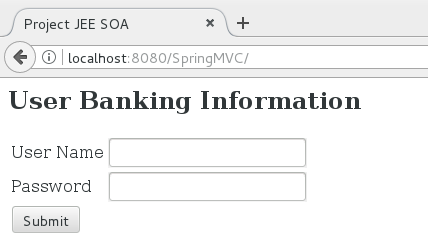
*com.projectJEE.common.controller*.



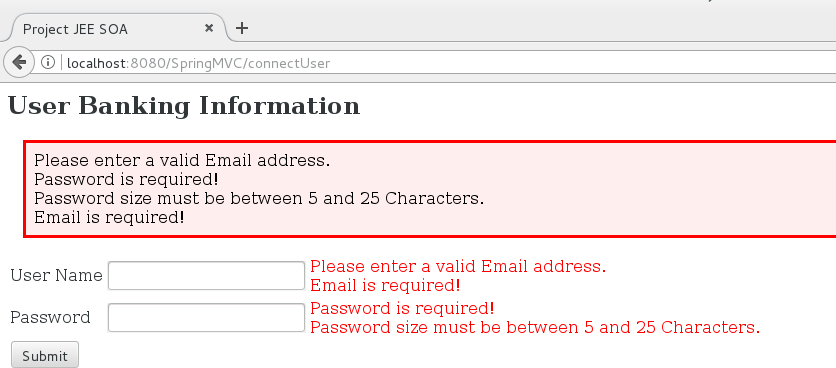


# Web Application: Functionality and User interface

## User logging screen



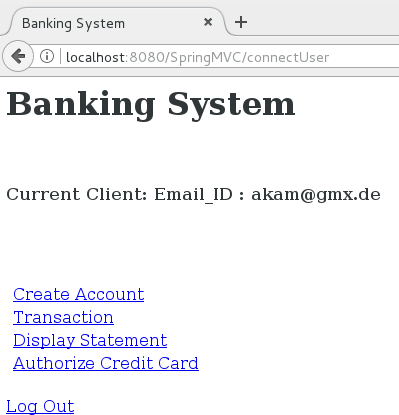
In this logging interface screen, the user is prompted to enter his credential, namely the user id and the Password. The user id is in the form of email pattern. The password is required and the size must be between 5 and 25 Characters. The validation is performed at the server-side using JSR 303 validation as shown bellow.



The credential are verified against the data stored in the Data-Base, in the table user\_pass. The password could be stored in the form of a MD5 or SHA-hash in case of data leakage.

## Main Menu screen

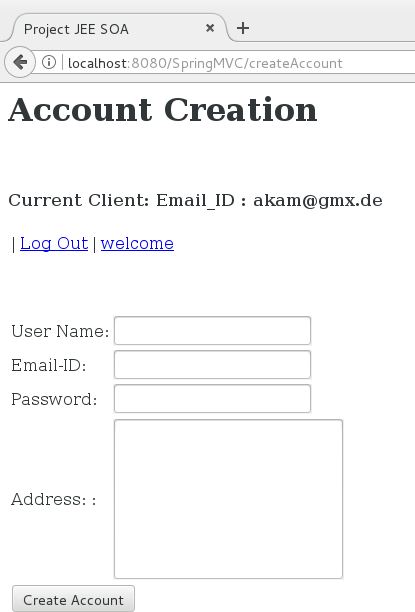
Once the user has entered the correct credential, he/she is presented with the main menu offering various banking operations such as account creation, money transfer via transaction, list of transactions filtered by date and so on.



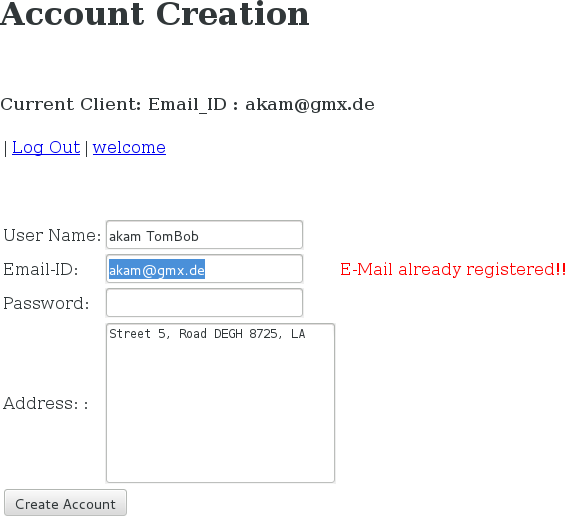
In case of incorrect credential, the user is presented with the user logging page again and prompted to enter a valid user id and password.

## Account Creation

The user id (email) and password are required for the creation of a new account.



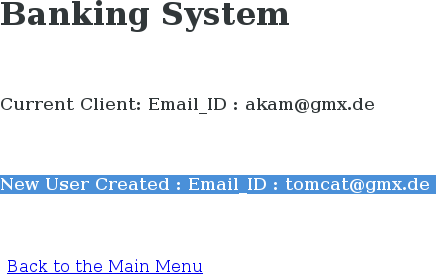


The Email\_ID field is checked against the database to verify if this email is already registered.



The table “*user\_passw*” before creating a new user.



After pressing/clicking the “Create Account” button, the user will be presented with the following confirmation screen.

Here a new account “[tomcat@gmx.de](mailto:tomcat@gmx.de)” has been created. This could be verified by listing the users in the table as shown bellow.



By default each new user created receives 50000 as current balance.

## Debit – Credit

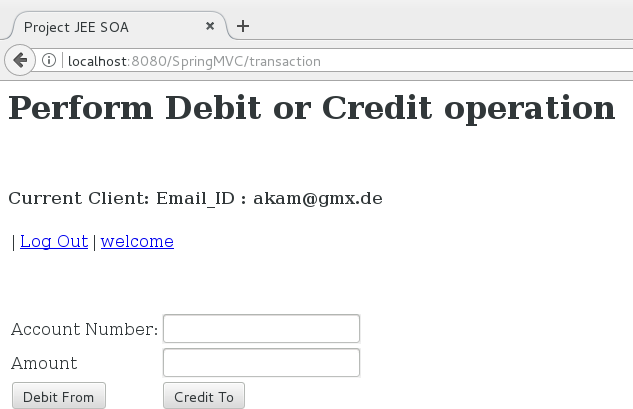
**Debit – Credit from the current account to another account (transactions)**

It is crucial for a banking system, to maintain the correct state of the data. When multiple related operations are performed on the database, and if at least one operation fails then it is required to bring the database to its original state (rollback). For example when we transfer the money from one account to the another account, in other words when we debit an account to credit another account, it is crucial to ensure that either both operations (debit and credit) are performed or neither of these operations are executed.

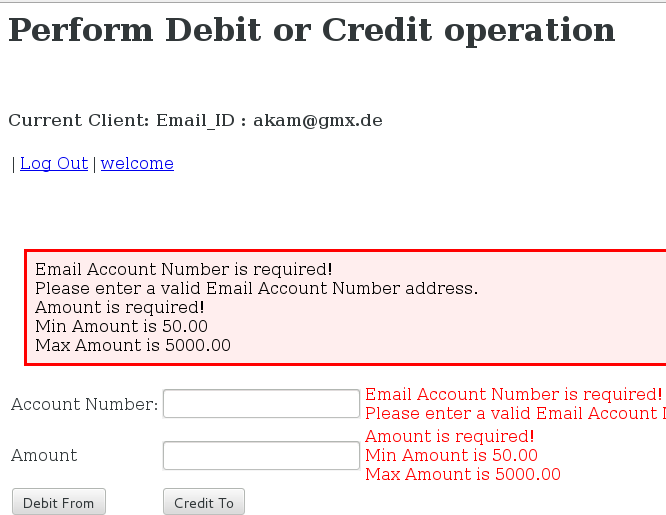
I have decided to use ***JDBC API*** which is the industry standard for database-independent connectivity The JDBC driver is provided by the database vendor and JDBC offers ***Transaction management***.

I have grouped all related instructions to the transaction in one ***Batch process*** and these operations will be executed as single operation. By using *Commit()*, changes done in the database will be made permanent. By using *RollBack()* changes done in the database will be removed.

Clicking the Transaction link in the main menu will bring the following screen interface where we can perform debit and credit operations. Two fields should be specified. The first field is the account number in the form of an Email\_ID. The second field is the amount to be debited or credited.



Here are the requirements definitions for the debit and credit operations.



The validation is performed at the server-side using JSR 303 validation as shown. The account number should follow an Email pattern and the amount filed should between 50 to 5000.



The account number is then checked against the database to verify the user validity.

If the account number (Email\_ID) specified in the first field is the same as the logged current user, then no operation will be performed and the application will ask the user to enter another account.



If the Account number (Email\_ID) is registered as a valid account in the Database and the amount is between 50 to 5000, then Debit and Credit operation could be performed .

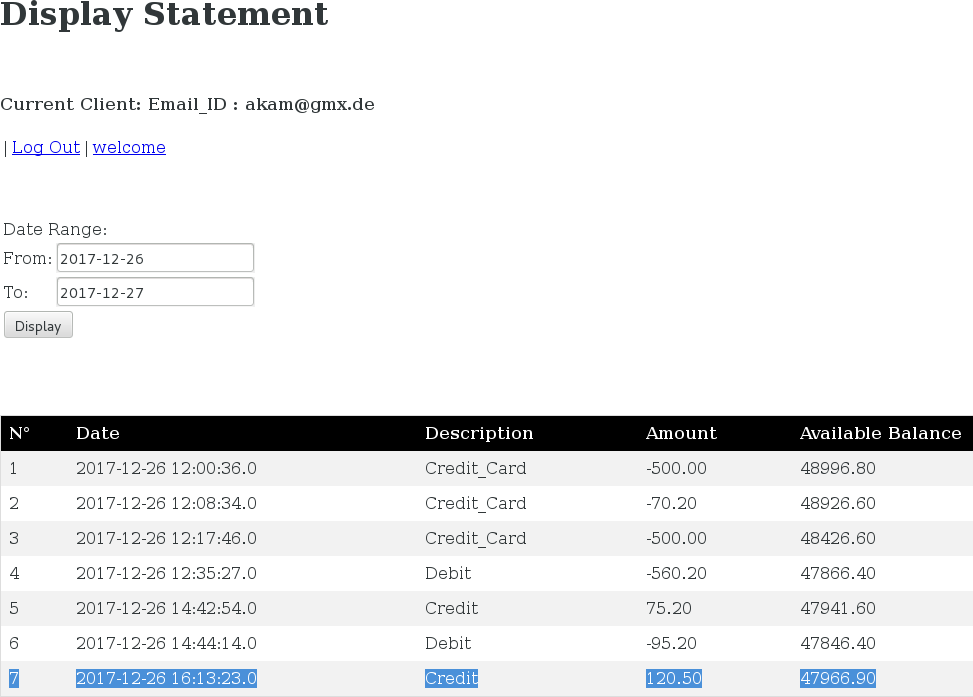
Pressing the "*Debit From*" button will debit the account specified in the first field and credit the current logged client.

Pressing the "*Credit To*" button, the account specified in the first field will be credited and the current logged account will be debited.

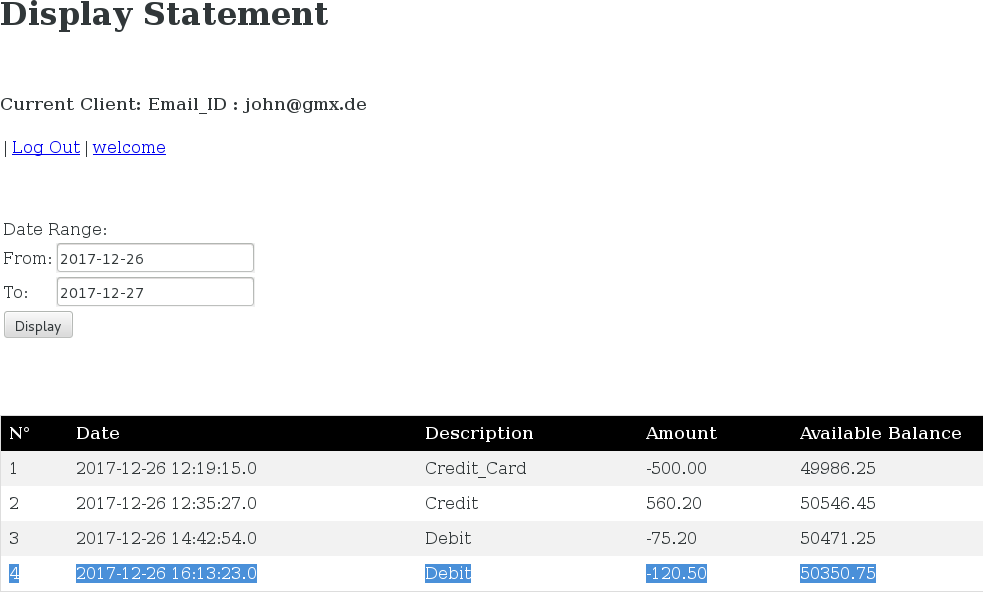
Below is an example where the “Debit From” button is pressed.

In this

example the account [john@dmx.de](mailto:john@dmx.de) is debited by an amount of 120.5 and the current account [akam@gmx.de](mailto:akam@gmx.de) is credited by same amount 120.5. These operations could be verified by listing the operations as shown bellow.



The current account, [akam@gmx.de](mailto:akam@gmx.de) has been credited by 120.50.

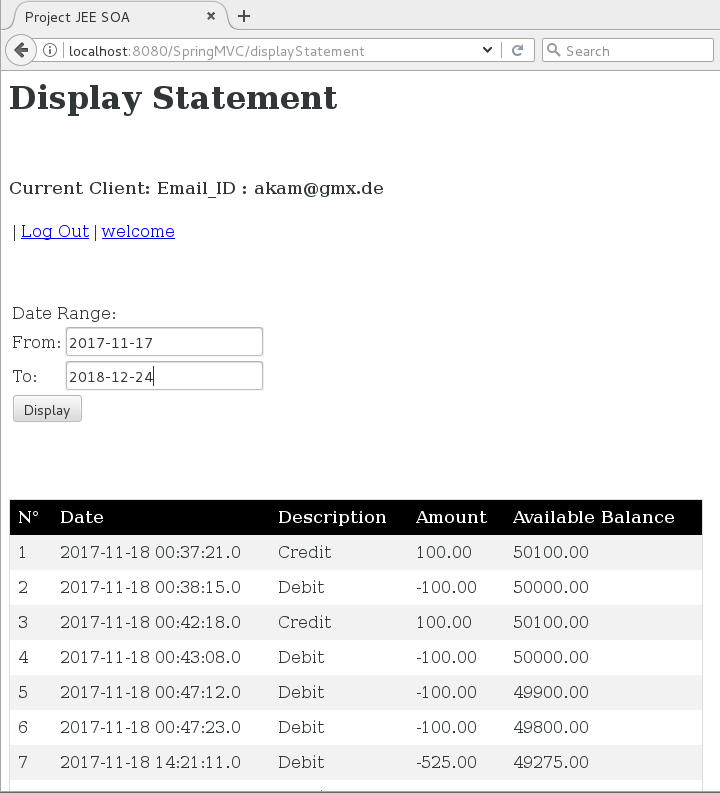
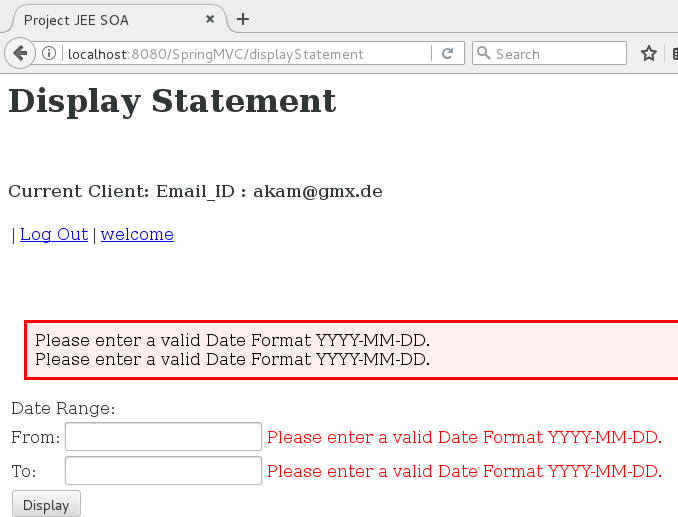
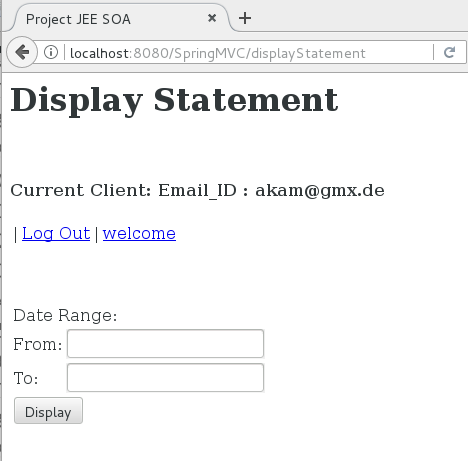


The specified account, [john@gmx.de](mailto:john@gmx.de) has been debited by 120.50.

*(\*) Notice that we could also define simpler requirements, where just the specified account number (Email\_ID) is either credited or debited.*

## Display Statement

**List of the transactions filtered by date for the current logged user.**

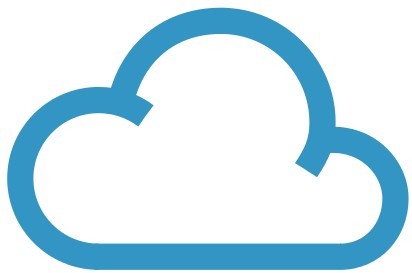


## Credit Card Transaction

Here are the requirements for the credit card transaction:

For the Credit Card transaction, a RESTful Web Service has been created to check the validity of the credit card, such as the Card number, CVV code, the name (card holder or email\_id). If the credit card details are found correct, then the transaction is accepted. JSON objects will be exchanged between the main banking system application and the Credit Card Web Service.

Jersey RESTful Web Services framework is used for developing the RESTful Web Services in Java. It provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.



RESTful Web Server

**Internet**

Credit Card

JEE JAX-RS

A POST request is sent to the web server with the Object bellow for verification:

{

"CreditCardNumber":"4012345678912124", "CC\_CVV":"349",

"CC\_Email":["a](mailto:akam@gmx.de)k[am@gmx.de](mailto:akam@gmx.de)", "CC\_validity":"false"

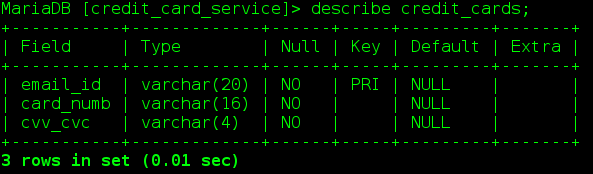
Data Base Server

MySQL

Client Consumer Post Request JSON Object Jersey Framework

}

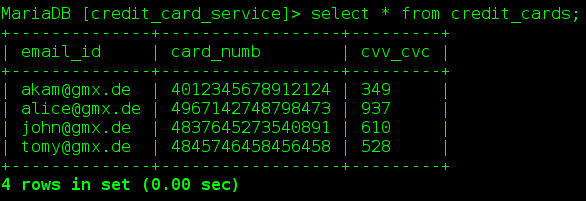
The RESTful web server returns the following Object after check:

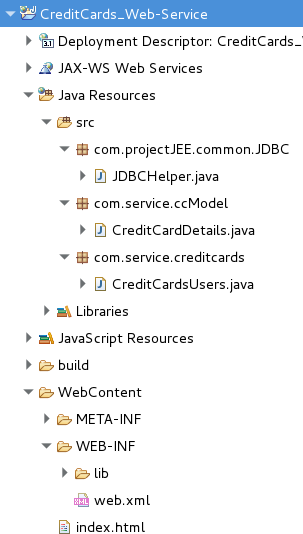
{

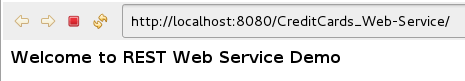
"CC\_CVV": "349",

"CC\_Email": ["aka](mailto:akam@gmx.de)m[@gmx.de](mailto:akam@gmx.de)", "CC\_validity": "true", "creditCardNumber": "4012345678912124"

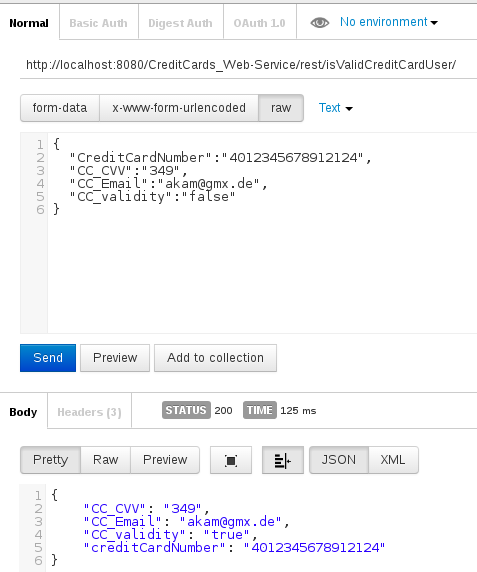
}



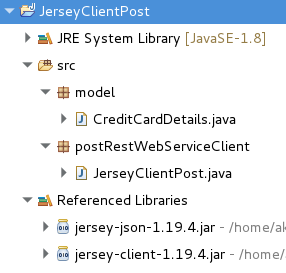
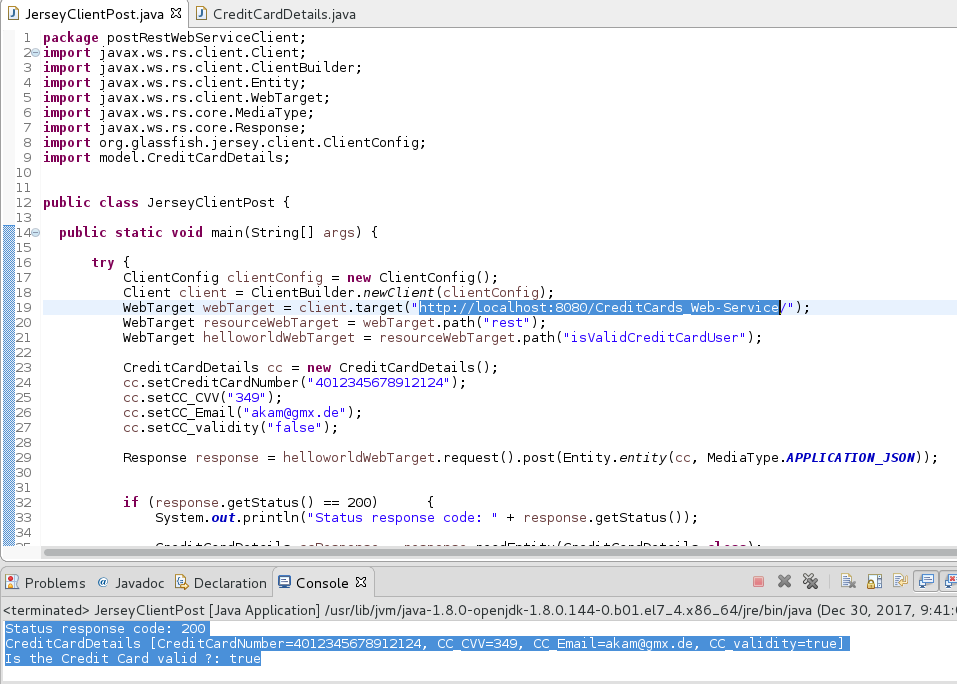
First start the RESTful Web Service

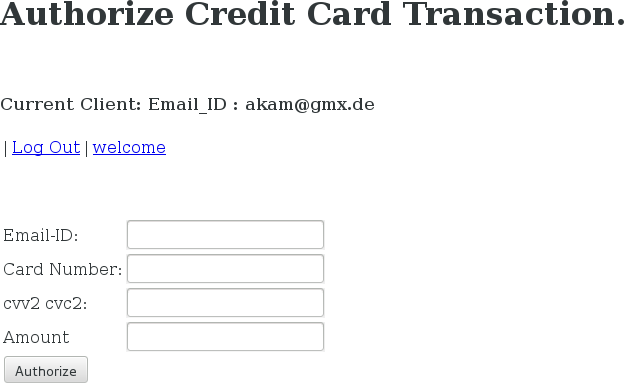


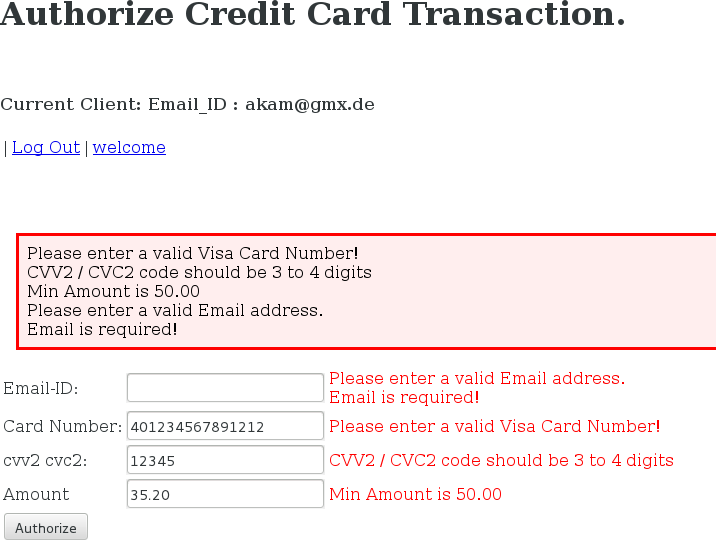
Next we can test the RESTful Web Service with REST Client tool like POSTMAN

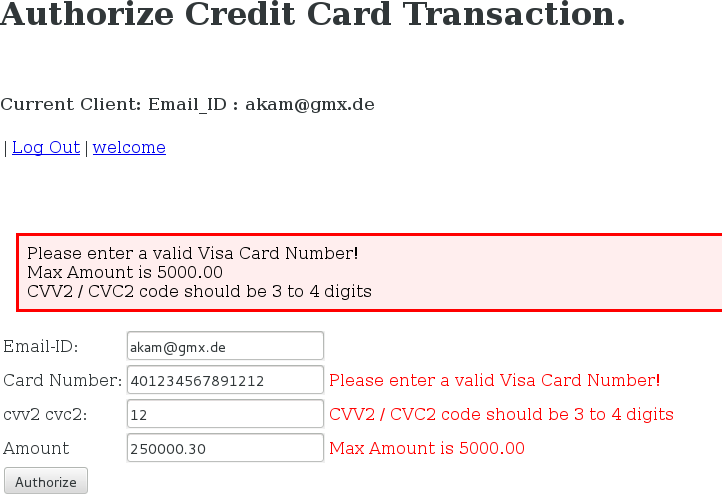


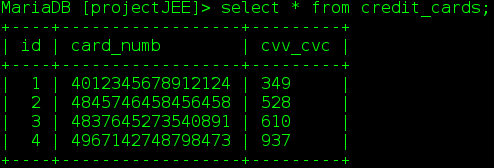
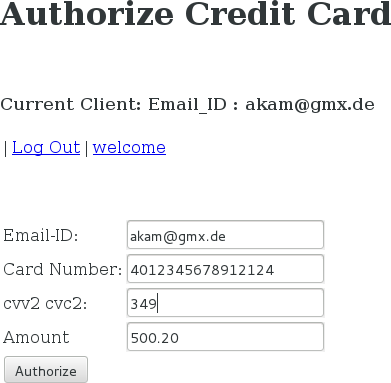
We can also develop our own RESTful client application as shown here bellow. The Jersey JAX-RS jars are require. The jersey-media-moxy jar file is also required. It allows us to convert POJOs to JSONs.







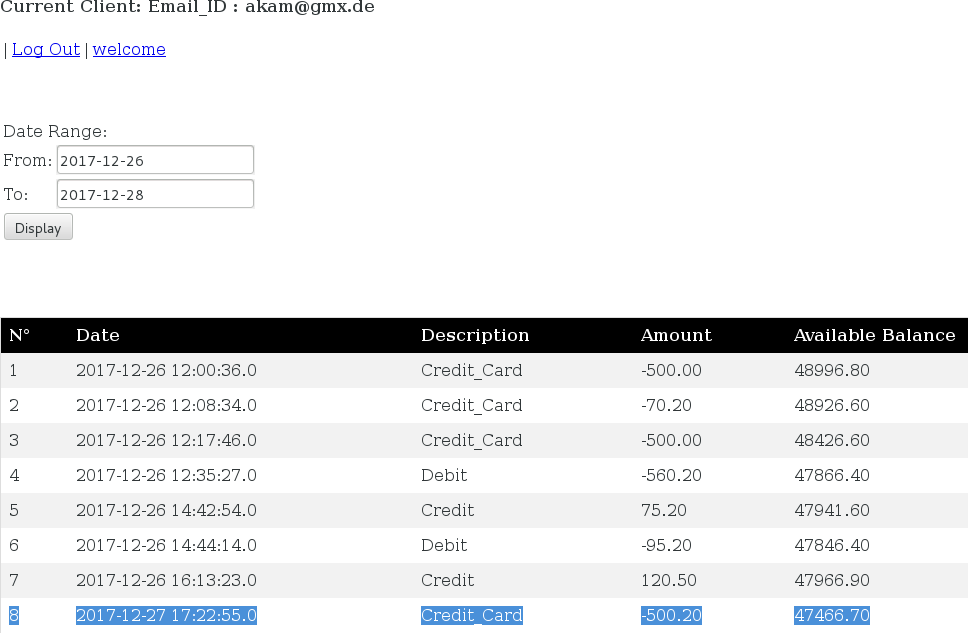




Above the user has entered a valid card number and the associated email account. So the user should be able to execute the transaction.

A green message will confirm to the user that the transaction has been executed as shown below.

Now we can list the transactions to verify that indeed the credit card transaction has been executed.



# Database

## JDBC (Java database connectivity)

I have decided to use ***JDBC (Java database connectivity)***, simply because I need more time to master Hibernate and integrated it efficiently with Spring MVC. This is more a JSE practice rather than JEE and as so, I still need to write lot of boilerplate code.

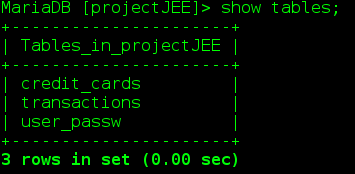
***JDBC API*** is the industry standard for database-independent connectivity and the driver is provided by the database vendor. JDBC offers Transaction management which crucial for a banking system (debit, credit, transfer from one account to another account). When multiple operations are performed on the database, and if one or more transaction fails then it is required to bring the database to the original state. To maintain the correct state of the data, transaction management is required. By using Commit(), changes done in the DB will be made permanent. By using RollBack() changes done in the DB will be removed.

#### Batch Processing

When required, to increase the performance, all queries are placed in one batch and is transferred to the Database as one batch and executed on the database.

***Stored procedures*** could also be used.

## DataBase Design, Tables



#### Database name: projectJEE

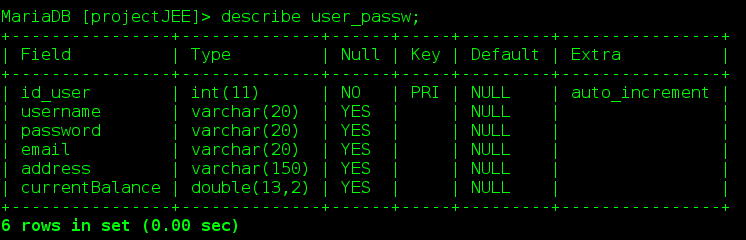
***3 tables*** are stored in the projectJEE data base:

#### user\_passw transactions credit\_cards

The database is designed to reduce the duplication of data and ensure referential integrity( 3NF Third normal form). 3NF is designed to improve database processing while minimizing storage costs.

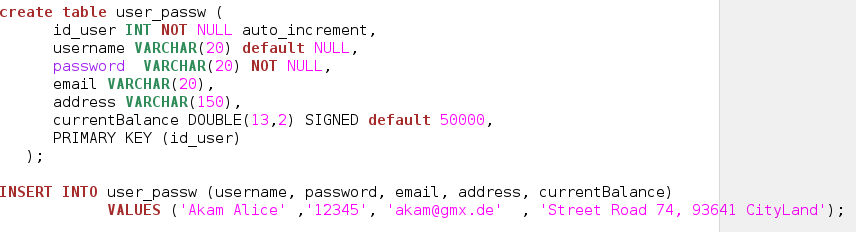
(\*) Note that the credit\_cards table in the projectJEE database was use in the first version of the

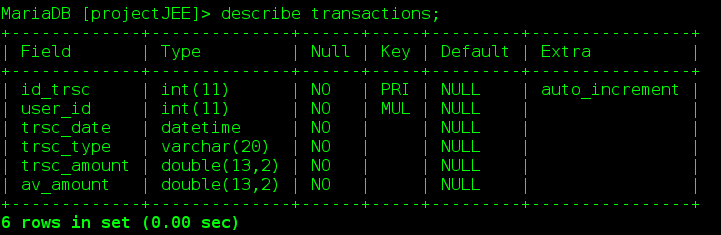
program, before implementing the RESTful Web Service.



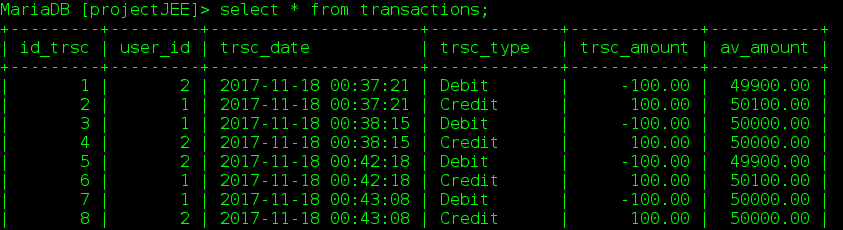
We can list the users in the table “user pass” table as shown below.

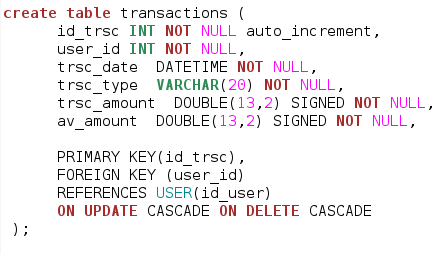


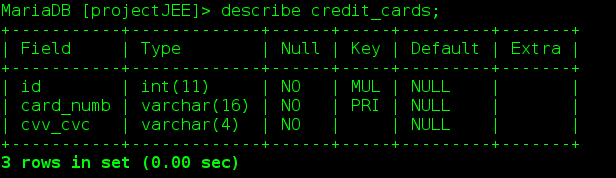




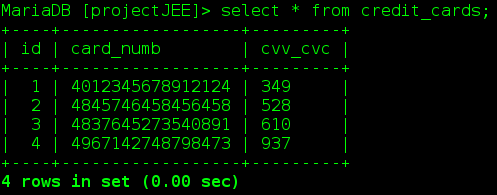
Here below are listed the transactions extracted from the “transactions” table.

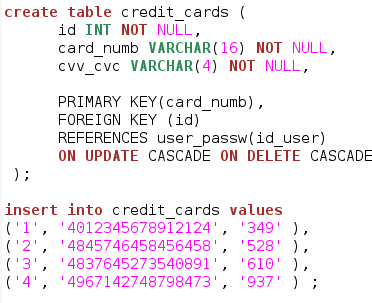






In the table *credit\_cards above*, the *card number* field is unique (visa pattern) and is used as primary key. The *id* field is a foreign key referencing the *id\_user* in the table u*ser\_passw* table. (onupdate delete cascade). Therefore a user could have many Credit Cards.

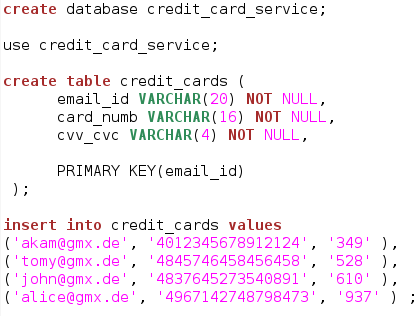
As shown below, the table credit\_cards contains the credit cards details associated with each user.

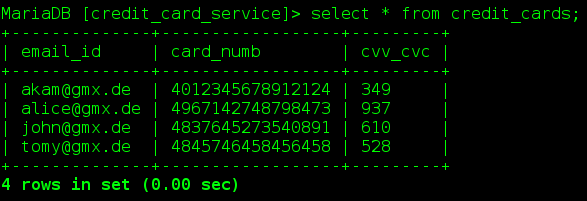
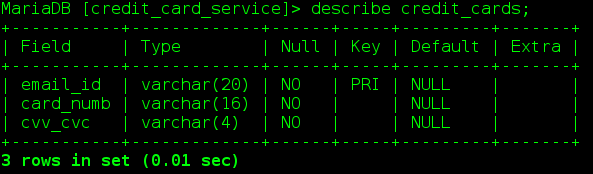


#### Database name: credit\_card\_service

***1 table*** is stored in this data base: ***credit\_cards***

This database is associated with the credit cards *RESTful Web Service*.





# 6 Bibliography

Java Enterprise Edition <http://www.oracle.com/technetwork/java/javaee/overview/index.html>

Apache Web Server <http://tomcat.apache.org/>

Maven Repository: <https://maven.apache.org/> <https://mvnrepository.com/>

MVC Spring Framework: <https://spring.io/>

DataBase: <https://mariadb.org/> <https://dev.mysql.com/>

RESTful Web Service <https://jersey.github.io/>

JSP Java Server Pages Standard Tag Library JSTL <http://www.oracle.com/technetwork/java/index-jsp-135995.html>