

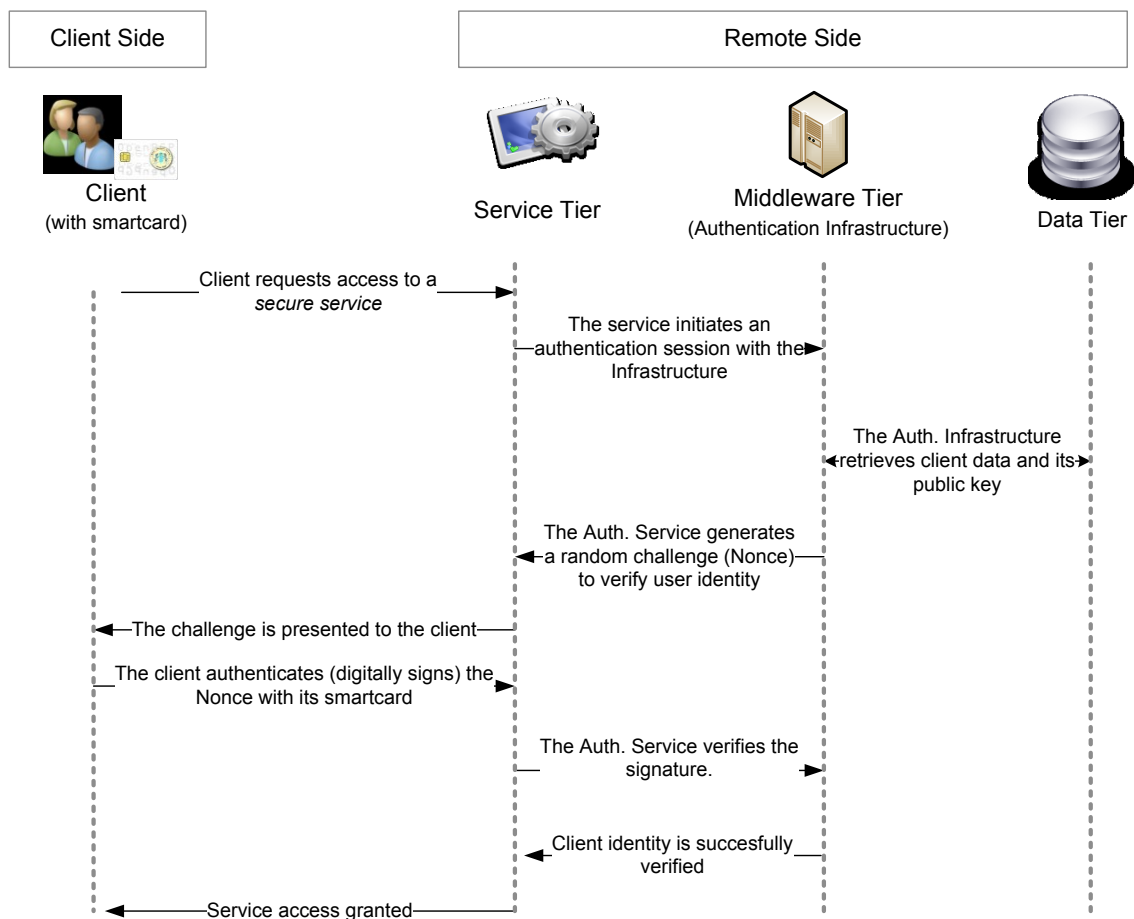
# A Simple Public Key Authentication Infrastructure based on the OpenPGP Smartcard and EJB 3

Primiano Tucci - <http://www.primianotucci.com>

## INTRODUCTION

User name and password combinations have typically been used to provide authentication to network resources. However, many users favor convenience to security, so they choose passwords that can be easily compromised. To address this issue, multifactor authentication uses a combination of components to provide secure access to network resources. These components include a device that the user has, a hardware token, and something the user knows, such as a Personal Identification Number (PIN). Smart cards are an increasingly popular form of multifactor authentication. This tutorial explains foundational concepts and provides a working implementation example of a smartcard-based authentication solution.

Figure below shows a typical interaction for a simple public key authentication session, based on challenge signature.



In this tutorial, we will show how to realize a very simple, but working, *Public Key Authentication Infrastructure*. Technical details of Nonce-based Public Key Authentication are omitted here, since a large amount of technology-specific tutorials can be easily found on the network. We will highlight how the proper usage of *component-driven development* can seamlessly integrate an apparently compound feature (the smartcard-based authentication) with a (sometimes-complex) existing service. In this tutorial, we will give the guidelines for development of a sample Public Key Authentication Infrastructure, basing on a typical application scenario:

- **A secure service** (e.g., an e-commerce site, a payment platform, an accounting zone...) that needs to authenticate clients and verify their identity.
- **An authentication tier** that will take care of the authentication process business.
- **A data tier** that will keep (among the other information probably needed by the *secure service*) the authentication data used for the client identification.
- **A remote client:** that accesses the secure service using a smartcard to prove its identity and authenticate with the service.

Indeed the ideas illustrated above are absolutely general and could apply to a wide variety of applications. In order to provide a concrete example we restrict the scope of this tutorial on the following technologies:

- **The OpenPGP Card:** The choice of the OpenPGP Card (rather than any other smartcard) is mainly due to the availability of open source drivers and java libraries (see <http://www.primianotucci.com/go/jopenpgpcard>). The same principles can be, of course, extended to other smartcards and secure tokens. Details of the OpenPGP are Card discussed later. Furthermore, the *JOpenPGPCard project* includes a sample applet ready for both standalone and web-based use that will manage the client interaction with the OpenPGP smartcard.
- **Java EE Compliant Application Server.** The business logic of the authentication infrastructure will be developed as an *Enterprise Java Bean*. In particular, in this tutorial, we will exploit the facilities introduced by the *Stateful Session Beans* and the *Java Persistence APIs*. The source code files provided along with this tutorial have been tested with the open-source Glassfish Application Server (<https://glassfish.dev.java.net/>). Other Java EE compliant application servers may work as well.
- **Bouncy Castle Java Crypto APIs.** The public-key verification requires a set of cryptographic libraries to achieve the required functionalities. We will use, in this example, the open-source Bouncy Castle APIs (<http://www.bouncycastle.org/>) for the digital signature verification process.

## THE OPENPGP SMART CARD



As almost any smartcard, the OpenPGP card consists in an hardware identification module with an embedded crypto processor and secure memory. Typically, an OpenPGP smartcard contains the following information:

- **Card Identifier:** a serial number hard-coded into the smart card
- **Card Holder Data:** Name, Surname...
- **Security PIN:** has to be entered each time we perform a security operation such as signature, authentication, and decryption.
- **Three 1024 bit RSA Keys:** A Signing Key, An Encryption Key and An Authentication Key

In this tutorial, we will focus our attention on the *Authentication Key*. The purpose of such security key is to take part in the *authentication* of an arbitrary random data, the *Nonce*, generated by the *Authentication Infrastructure* in order to testify the identity of the owner. Once the client's smart card has signed the *Nonce* using its *private* (Authentication) key, the *Infrastructure* can verify the validity of its signature against the client *Public Key*, previously registered into the infrastructure database. (See [http://en.wikipedia.org/wiki/Public\\_key\\_cryptography](http://en.wikipedia.org/wiki/Public_key_cryptography) for major details)

#### THE BACKEND DATASET

First, we will need a backend dataset where the public information of the clients will be kept, in order to afterwards retrieve information related to the authenticating client and logically associate a smartcard to its owner. For each registered smartcard, we will keep track of the following information:

Card ID 🗝	Username	Public Key
01010001AA	John	-----BEGIN PUBLIC KEY----- Public Authentication Key data of John's smartcard -----END PUBLIC KEY-----
01010001BB	Jack	-----BEGIN PUBLIC KEY----- Public Authentication Key data of Jack's smartcard -----END PUBLIC KEY-----
01010001CC	Fred	-----BEGIN PUBLIC KEY----- Public Authentication Key data of Fred's smartcard -----END PUBLIC KEY-----

**Card ID:** needed to univocally look-up the smartcard in the database.

**Username:** the friendly-name of the client that owns the smartcard, will be used by the requesting service to extract additional context information (customer address ...)

**Public Key:** The public key data, which corresponds to the client's smartcard Authentication Key, needed by the

Infrastructure for the signature verification process.

We would not provide, in this tutorial, a graphical administration area for the smartcard registration and database maintenance. In order to get the example to work, smartcards have to be manually registered into the database, according to the above table (the schema of the above table is included in the source code archives). The PEM-Encoded public keys can be extracted using the *JOpenPGP Card Editor* tool (see <http://www.primianotucci.com/go/jopenpgpcard>).

In production environments, the *JOpenPGP Card driver* could be further exploited in order to provide an automated service that extracts, upon client registration, the public key from its OpenPGP card and automatically stores back into the database, along with the other application-specific client information required. An example of public key extraction using a web-based applet can be found in the source code of another project based on the OpenPGP Smartcard and the *JOpenPGP Card driver* (see source code of <http://dev.primianotucci.com/openid/>)

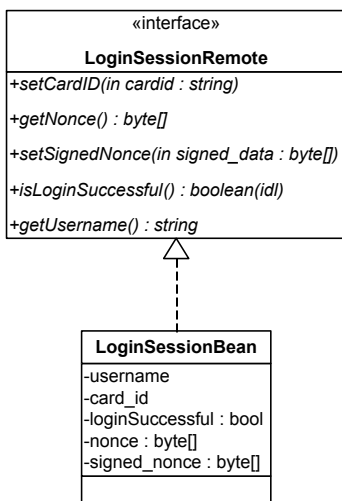
#### AUTHENTICATION INFRASTRUCTURE: THE MIDDLEWARE AUTHENTICATION SERVICE

The core of the authentication process is managed by what we called, in this tutorial, the *Authentication Infrastructure*. The purpose of such entity is to enclose, in a reusable and independent component, the business intelligence of the smartcard-based authentication process. Once we clearly define the component target and requirements, we can isolate them from the application-specific context, developing an autonomous business component.

The source code files provided [[Download](#)] contains three java projects (created with the [Netbeans Platform](#)) organized as follows:

- **/EJBSmartcardLogin-ejb**: project containing the EJB Bean of the *Authentication Infrastructure*
- **/EJBSmartcardLogin-app-client**: simple standalone client that uses the former EJB to authenticate the client.
- **/WebSmartcardLogin**: web-based example (JSP presentation + servlet controller) that shows how the same EJB can be exploited for web client authentication.

The *Authentication Infrastructure* is organized as follows: basing on the [Enterprise Java Beans technology](#) we develop a *Stateful Session Bean*, in this example the *LoginSessionBean* class. Such component would manage the entire authentication process of a remote client, providing primitives for the public key, challenge-based, smartcard authentication. The state of our *LoginSessionBean* will keep track of the proceedings of the login process along the whole authentication procedure.

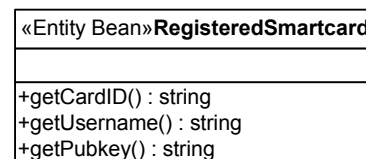


The component life cycle is arranged as follows:

- Upon initialization, the *LoginSessionBean* will generate a random challenge (the Nonce) and store it in a local field.
- The secure service retrieves the Nonce (through the `getNonce()` getter) and sends it back to the authenticating client.
- The client authenticates Nonce with its smartcard (the smartcard operates a digital signature of the Nonce using the *Authentication Key*) and sends back the *signed data*, along with its *Card ID*
- On the basis of the client's *Card ID*, the *LoginSessionBean* looks-up the client data from the database and verifies the signed data against the client public key. The result of this verification is stored back into the *loginSuccessful* field.
- The *secure service* checks whether the authentication has been successfully completed (via the `getLoginSuccessful()` getter) and proceeds furnishing its services to the requesting client (or reporting an error message in case of failure)

#### DATABASE INTRACTION: THE JAVA PERSISTENCE APIS

The database look-up operation is relegated to the [Java Persistence APIs](#). For this purpose, we introduce the *RegisteredSmartcard* Entity Bean, mapping its fields to the dataset structure. Retrieval of smartcard information is done by the *EntityManager* (part of the JPA APIs) , having set the *CardID* field as primary key of the dataset. The backend data source (in this example we defined a sample data source in the Application Server) is defined in the *persistence.xml*, along with the mapping of the *RegisteredSmartcard* Entity Bean (take a look at [this tutorial](#) for instructions on how to configure a custom data source in the Application Server)

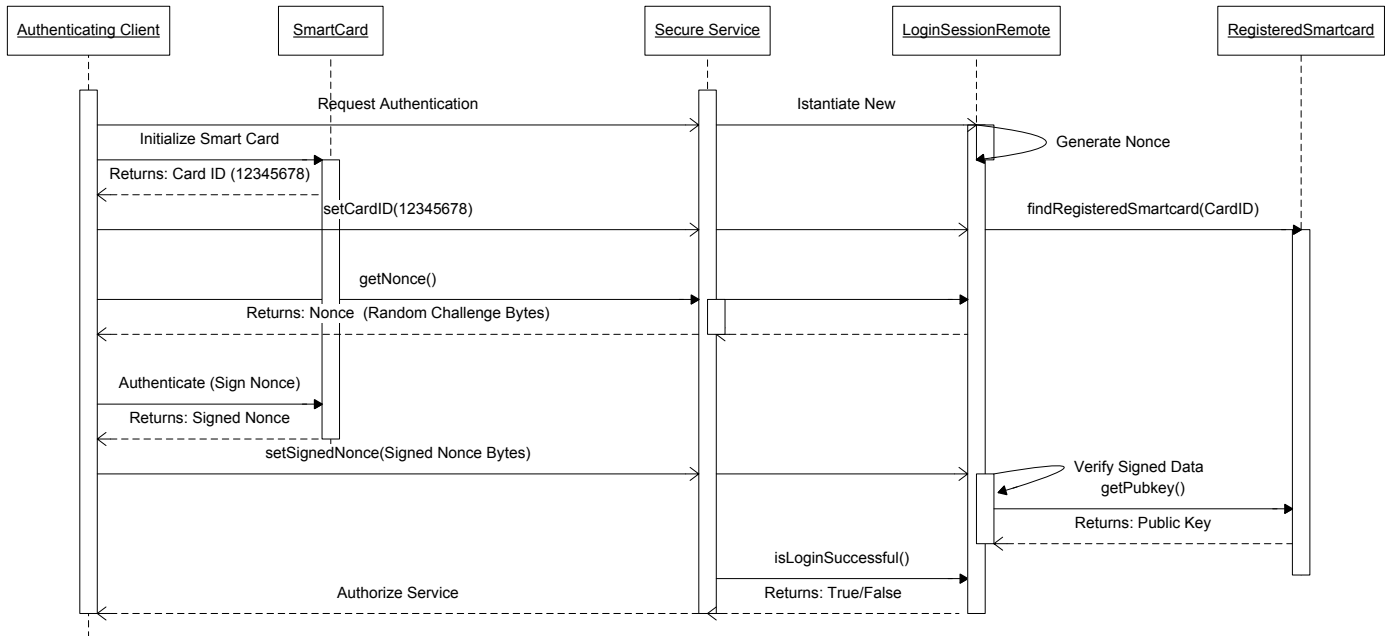


```

<persistence-unit name="EJBSmartcardLogin-ejbPU" transaction-type="JTA">
  <jta-data-source>jdbc/sample</jta-data-source>
  <class>com.primianotucci.smartcard.ejb.RegisteredSmartcard</class>
</persistence-unit>

```

Below we report the complete sequence diagram that highlights the interaction between the *Client*, its smartcard, the *secure service*, the *authentication infrastructure* and the *data tier*.

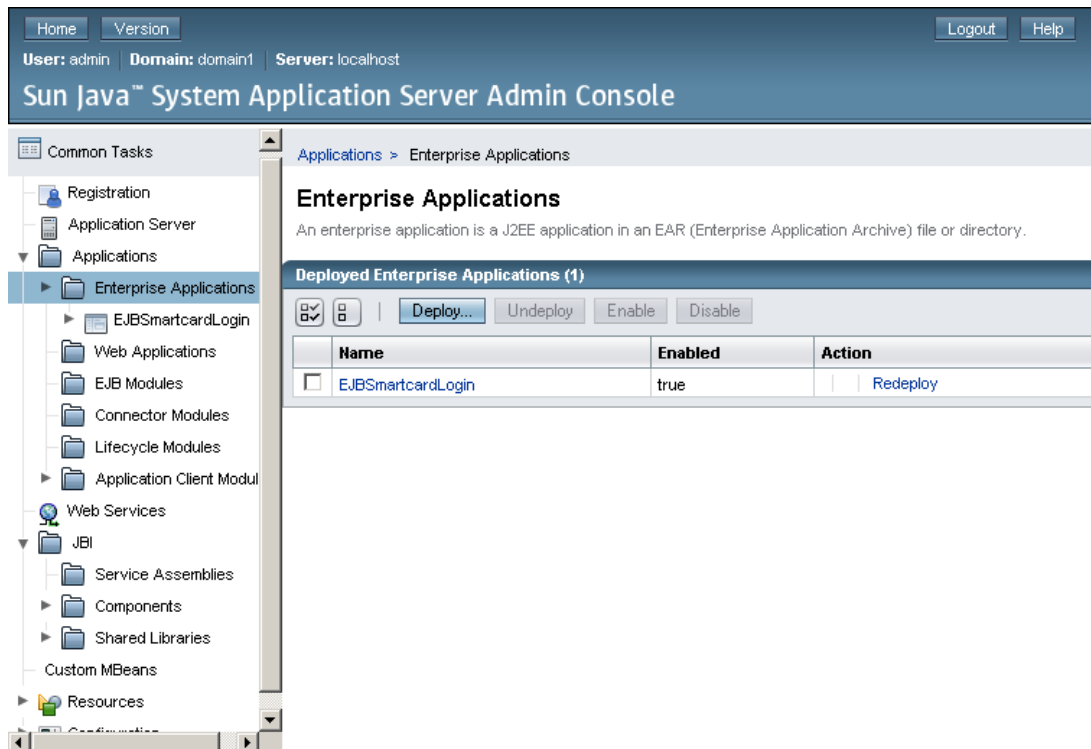


## RUNNING THE EXAMPLE

Along with the source code files [[Download](#)] is provided a compiled *Enterprise Application Archive* (EJBSmartcardLogin.ear file under the dist/ folder). The application contains the binary version of all the three projects discussed above. The instructions below will show how to deploy and run the examples in a Glassfish Application Server (v2ur2).

1. Start your glassfish server ☺
2. Point your browser at the address <http://localhost:4848/> enter your administration credentials (default values for Glassfish server are user:admin, password:adminadmin) and open the Application Server Admin Console
3. Navigate to the panel Resources / JDBC / JDBC Resources and create a suitable JDBC resource. The default JNDI name used in the example is: jdbc/sample. Probably your copy of Glassfish Server would already have such resource. In this case you have to just import the schema (if you use Netbeans Platform you can directly import the schema file located, in the source archive, under folder /EJBSmartcardLogin-ejb/src/conf/ APP\_EJBSmartcardLogin-ejb.dbschema
4. Add the smartcard entries to the database, following the structure of Page 3. Public keys can be extracted from the smartcard using the JOpenPGP Card Editor [[Launch it via Java WebStart](#)]

- Copy the EJBSmartcardLogin.ear archive into the /autodeploy directory of your domain root (e.g. [Glassfish\_Installation\_Root]/domains/domain1/autodeploy/. The Enterprise Java Bean project and the client projects will be deployed in a few seconds.
- If you now take a look at the page: Applications / Enterprise Applications. You will see a row (see figure below) that correctly reports the deployed *EJBSmartcardLogin* application




- Point your web browser at the address <http://localhost:8080/WebSmartcardLogin>. You will be promptly redirected to a login page. An applet (see figure below) will prompt you to insert the OpenPGP card into the reader. Insert your card, type your PIN and click the Login button.

## New Login

**Nonce Challenge** : 5676da23fe3f916aef6e22732d293fc37c4db875

JOpenPGPCardLogin Applet - Primiano Tucci

Reader:  

Status:

PIN:

- The controller servlet will redirect you to a simple JSP page that will report the result of the authentication process plus a set of diagnostic information (for test purposes only). In the case of a successful login the authentication page will report the username that corresponds, in the database, to the authenticating smartcard.

## Authentication

**Signing Nonce Was :** 5676da23fe3f916aef6e22732d293fc37cdb875

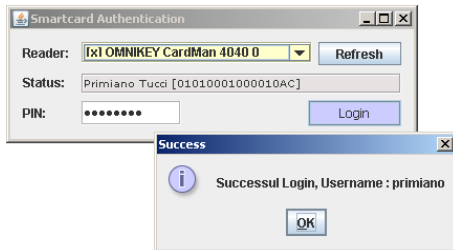
**Signed Data is :**

40f255de7d134110941f374876d19eebced5dea370c313357339afe763771e183742c464

**Card ID is :** 01010001000010AC

**Authentication:** Success (User: primiano)

9. If you wish to try the standalone client, instead of the web-based one, you can launch the



EJBSmartcardLogin-app-client.jar WebStart from the Application Server Admin Console (available under /Applications/Enterprise Applications/ EJBSmartcardLogin), Sub-Components page. The standalone works with the same logic as the web-based application, with the only difference that the service logic is integrated, this time into the client application rather than in a servlet.