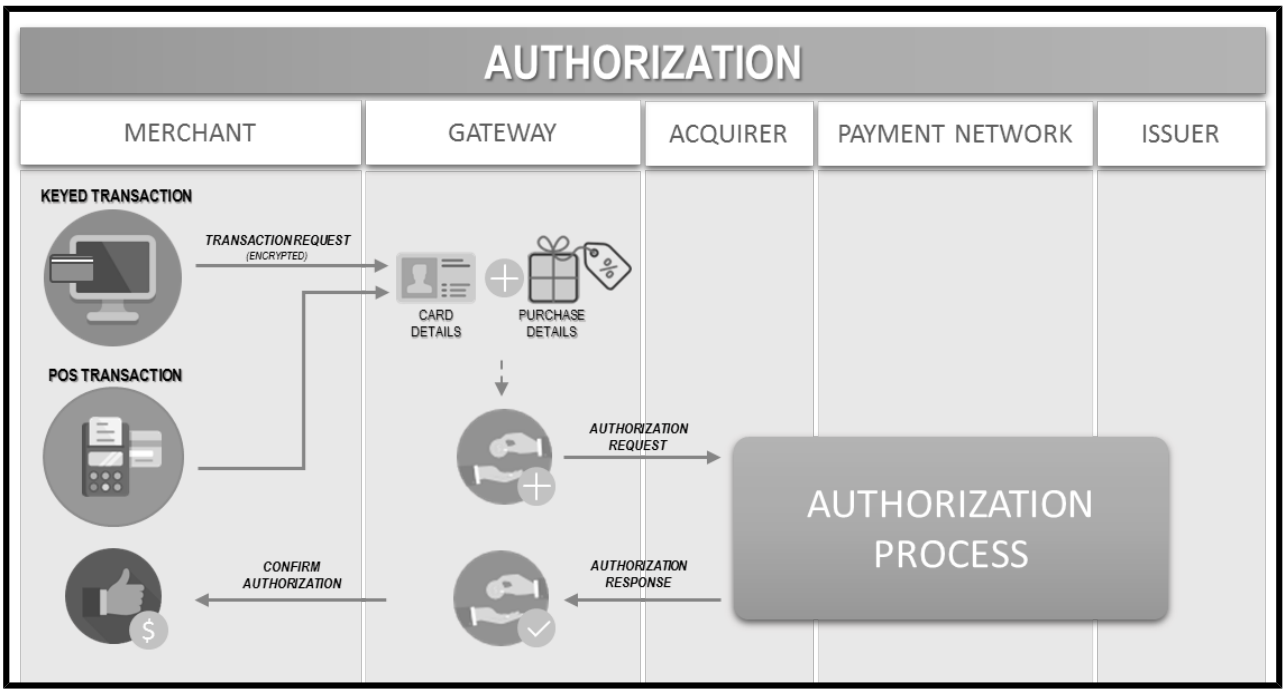
Mobile SDK - Secure Card Integration Guide

Table of Contents

1 Introduction

Secure Card is a functionality built to provide more security and agility for merchant's clients during payment transactions, but to understand it we need to start talking about a normal **Authorization Transaction Flow.**

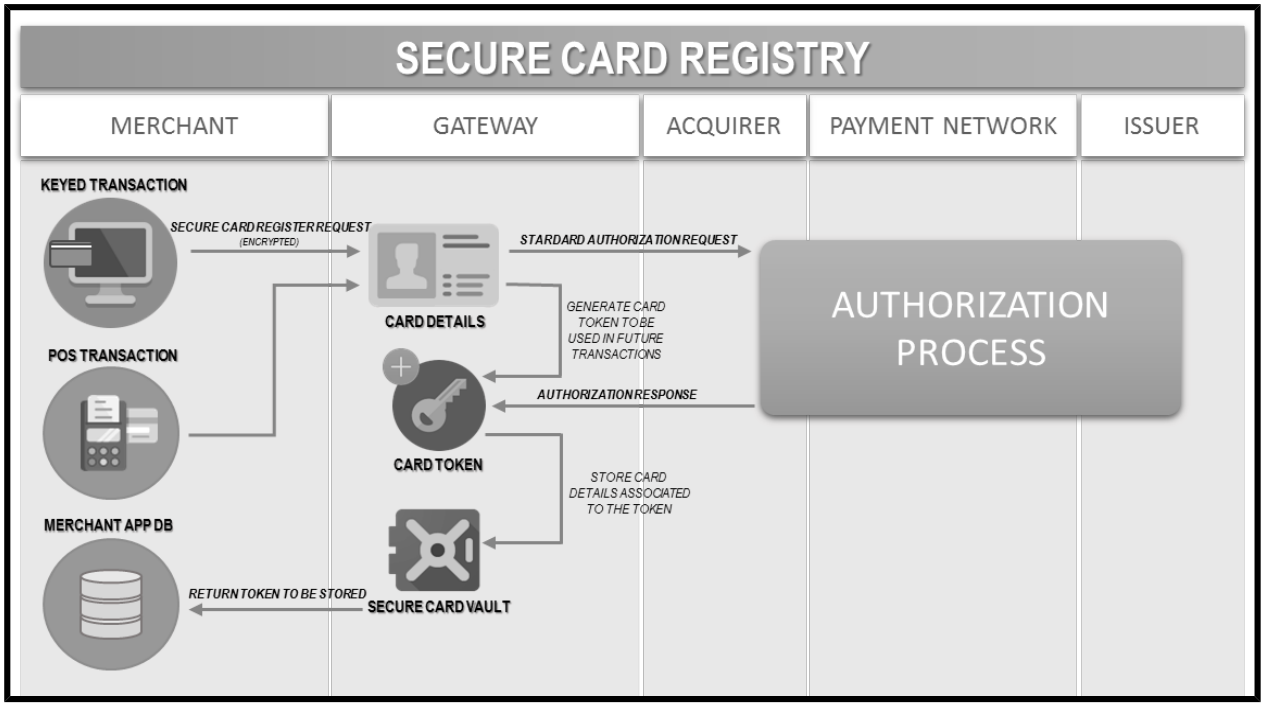


Considering the transaction shown, we can see that constantly sensitive data about the merchant's client, like the Card Details, are exchanged between the parts. Because of that, the parts involved always need to define rules and adopt solutions that provide more security for this exchange and preferable, reduce sensitive data traffic. Another question raised from this example is that every time an authorization is executed it's necessary to input all the card data again.

The Secure Card Feature was developed to address this issue, but how exactly are your solutions helped by that?

The Secure Card is a feature divided basically in two parts: the maintenance of tokens, that represent card identities (or Secure Cards), and the use of those tokens to execute payment transactions.

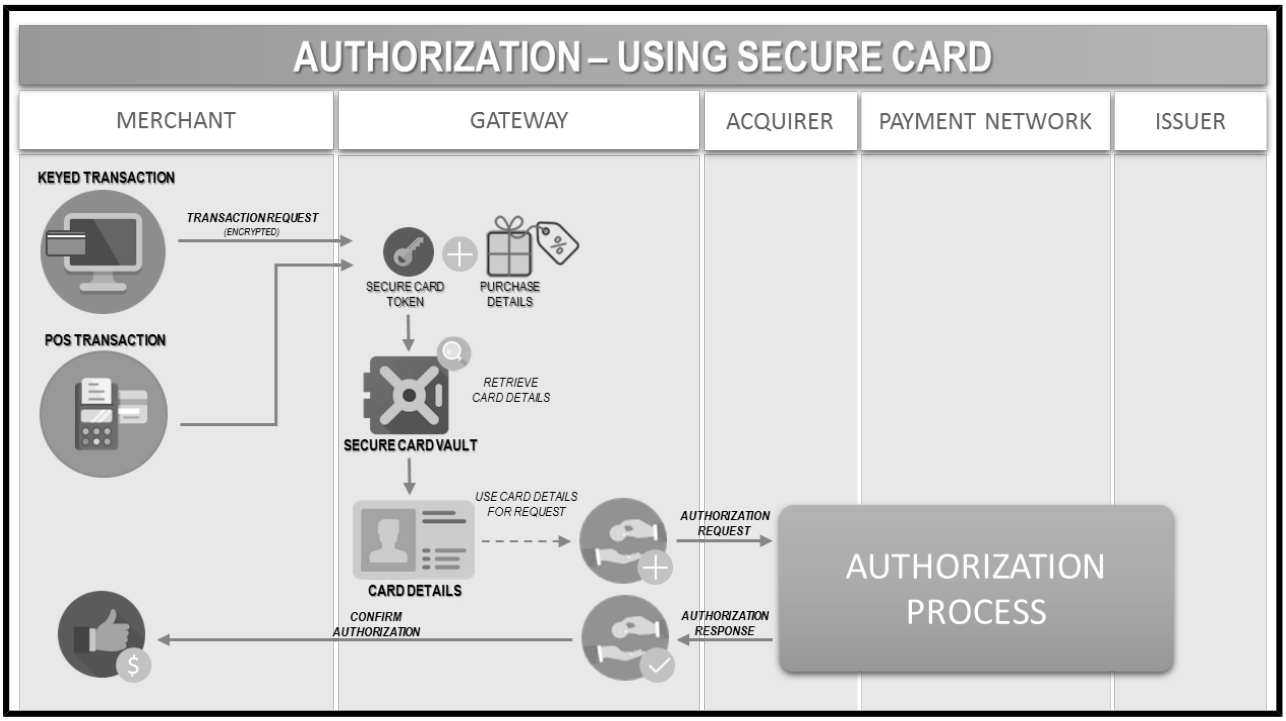
The registration can occur in two ways: during an authorization transaction, you can register the card's data, already informed for the transaction, as a Secure Card; or simply, you can register a card as a new Secure Card, but we are going to see more details in the second part of this guide. In the next figure we explore just the registration of a Secure Card example.



- First the card details are inserted – in a Keyed way, where you submit all the card's data on a form, or using a **POS**, where the data is read by the card reader device.
- Depending on the terminal's configuration, the card data and the acquirer rules, the Gateway can send a standard authorization transaction to try to validate the card, using its **CVV**, so the next authorizations done by using the Secure Card become easier.
- Then, The Gateway generates the card's Token and stores it for future use.
- Finally, the token is passed back to the merchant, that needs to associate, somehow, the card details first informed to the token returned as a response.

Note: In case you are asking yourself ,"*what's going to happen to the transactions used to validated card?*", relax! The transactions executed to validate the cards' details are automatically voided to guarantee that none undue transaction is processed by the system.

From this point on, every time the merchant executes a new transaction, instead of using the card's data, the merchant just need to send the token.



Note: Until the date and time of present guide's release, only the Secure Card Maintenance part of the feature was available for the Mobile SDK – that means Register (during an authorization or not), Update and Delete – and for that reason, it's the part that represents the focus of this guide. The use of the Mobile SDK to implement authorizations, using a Secure Card, isn't still available.

2 Implementing the Features

Each platform (Android and IOS) has a set of configurations and implementations necessary that need to be done before using the Mobile Secure Card Feature.

You can see more details about those configurations and implementations at the links provided in **Appendix B – Useful References** (SDK – Basic Configuration and Implementation to use the SDK).

So, if you **ALREADY** have done that, or **AFTER** you do it, jump to the right subsection, depending on your platform: **2.1 Android Implementation** or **2.2 IOS Implementation**.

These subsections will describe the necessary details to implement the business calls and get the responses for the Secure Cards feature, as:

- **Secure Card Registration:** a common registration of a card as a Secure Card.

- **Secure Card Update:** the update of card details of a registered Secure Card.

- **Secure Card Exclusion:** to eliminate a card's Secure Card, disabling new transactions with that Secure Card.

- **Secure Card Registration by Sale:** when you want to generate a Secure Card for the card which had its data informed/ captured during a sale transaction.

Note 1: The online documentation about the CoreAPIListener interface – and all the other classes and implementation of the SDK – shown at the website doesn't have the implementation of Secure Card features, once those are specifically made for the clients which will receive this guide. The updated version of the CoreAPIListener interface and other references can be consulted at **Appendix A – Mobile SDK Code Samples**.

Note 2: Don't forget to request your Mobile SDK package version for Secure Card from our support team. This is going to be the first element you are going to need in order to implement anything.

Android Implementation

2.1 Secure Card Registration

In this implementation your application will have basically 3 (three) options for registering the Secure Card:

- **Registration without a card reader device – Keyed:** in this case, your application is going to ask to its User the card's details and, if it's necessary to your application, the owner's details.

- **Registration without a card reader device – OCR:** in this case, your application is going to provide the card's details, captured by a OCR solution, except the CVV, and if it's necessary to your application, the owner's details.

- **Registration using a card reader device – MSR or EMV:** in this case you application just worries about providing the card owner's details, if desired, once the SDK encapsulates all the logics involved in capturing the data from the card reader, being MSR or EMV.

2.1.1 Registration Without a Card Reader Device – Keyed

1. Capture the customer's data in your application – you decide the way.
    a) Card details: type; number; expiry date; card holder's name; CVV.
    b) Personal details: e-mail; mobile number; city; region; country; IP address.
2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard coreSecureCard = new CoreSecureCard();

// Customer Information
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Create a **CoreKeyedSecureCard** and set the card's data.

```
// Card's Data Method - Keyed
CoreKeyedSecureCard coreKeyedSecureCard = new CoreKeyedSecureCard();

// Card's Information
coreKeyedSecureCard.setCardType(cardType);
coreKeyedSecureCard.setCardNumber(cardNumber);
coreKeyedSecureCard.setExpiryDate(cardExpiryDate);
coreKeyedSecureCard.setCardHolderName(cardHolderName);
coreKeyedSecureCard.setCvv(cardCVV);
```

4. Create a **CoreTokenMethod** and set the **CoreKeyedSecureCard,** created before**,** on it. After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Token Method
CoreTokenMethod tokenMethod = new CoreTokenMethod(coreKeyedSecureCard);

// Method Information
coreSecureCard.setCoreTokenMethod(tokenMethod);
```

5. Set the *merchantReference* to the **CoreSecureCard**.

```
// Merchant's Reference for the Secure Card
coreSecureCard.setMerchantReference(merchantReference);
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

6. Associate **CoreCustomField** to your **SecureCard**.

```
// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

7. Call the Terminal's method to finalize the registration.

```
terminal.registerSecureCard(coreSecureCard);
```

8. Give the proper treatment to each possible response:
   a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.1 Secure Card Registration** subsection).
   b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.1.2 Registration Without a Card Reader Device – OCR

1. Capture the customer's data in your application – you decide the way, although using the OCR implementation comes from the assumption that your application is going to use an OCR solution for reading the Card Details.
    a) Card details: type; number; expiry date; card holder's name.
    b) Personal details: e-mail; mobile number; city; region; country; IP address.
2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard coreSecureCard = new CoreSecureCard();

// Customer Information
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Create a **CoreOCRSecureCard** and set the card's data.

```
// Card's Data Method - OCR
CoreOCRSecureCard coreOCRSecureCard = new CoreOCRSecureCard();

// Card's Information
coreOCRSecureCard.setCardType(cardType);
coreOCRSecureCard.setCardNumber(cardNumber);
coreOCRSecureCard.setExpiryDate(cardExpiryDate);
coreOCRSecureCard.setCardHolderName(cardHolderName);
```

4. Create a **CoreTokenMethod** and set the **CoreOCRSecureCard, created before, on it.** After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Token Method
CoreTokenMethod tokenMethod = new CoreTokenMethod(coreOCRSecureCard);

// Method Information
coreSecureCard.setCoreTokenMethod(tokenMethod);
```

5. Set the *merchantReference* to the **CoreSecureCard**.

```
// Merchant's Reference for the Secure Card
coreSecureCard.setMerchantReference(merchantReference);
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** token (**CoreSecureCardResponse.**secureCardReference) that you may receive as a response.

6. Associate **CoreCustomField** to your **SecureCard**.

```
// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

7. Call the Terminal's method to finalize the registration.

```
terminal.registerSecureCard(coreSecureCard);
```

8. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.1 Secure Card Registration** subsection).
    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.1.3 Registration Using a Card Reader Device – MSR or EMV

1. Capture the customer's data in your application – you decide the way – except for the card's data:
    a) Personal details: e-mail; mobile number; city; region; country; IP address.
2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard coreSecureCard = new CoreSecureCard();

// Customer Information
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Set the *merchantReference* to the **CoreSecureCard**.

```
// Merchant's Reference for the Secure Card
coreSecureCard.setMerchantReference(merchantReference);
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

4. Associate **CoreCustomField** to your **SecureCard**.

```
// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

5. Call the Terminal's method to finalize the registration.

```
terminal.registerSecureCard(coreSecureCard);
```

6. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.1 Secure Card Registration** subsection).
    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection).

Note: Maybe you didn't notice, but in this subsection, until here, we didn't talk about getting the Card's data. That's because, when trying to implement the **MSR** and **EMV**, you don't need to add any information regarding the Card's data (like the **CoreTokenMethod** you did for **Keyed** and **OCR**). Your Application just needs to pass the **CoreSecureCard** to the **Terminal** method and the SDK will realize that the data was not inserted on a **Keyed** or **OCR** way, and then the SDK will communicate with the Card Reader Device to get the Card's Data, but that is transparent to Your Application, so don't worry. Just make sure that there's a Card Reader Device connected before submitting the **CoreSecureCard** to the **Terminal** method, or you are going to receive an error message saying that you need to connect first, before any reading be done by the Device.

2.2 Secure Card Update

In this implementation your application will have basically 2 (two) options for updating the Secure Card:

- **Update without a card reader device – Keyed or OCR:** in this case, your application is going to retrieve a Secure Card, change any data informed and submit the data again for update, while there's no card device reader connected.

- **Registration using a card reader device – MSR or EMV:** in this case, your application is going to retrieve a Secure Card, change any data informed, except for the card's data, and submit the data, once the SDK encapsulates all the logics involved in capturing the data from the card reader, being MSR or EMV.

Note: Be aware of the main data you need to inform before updating the Secure Card. The CoreSecureCardResponse retrieved contains the whole structure in CoreSecureCardResponse.getCoreSecureCard(). You can use this same object to submit the changes and just adjust the data, depending on what Your Application's User changes.

2.2.1 Registration Without a Card Reader Device – Keyed or OCR

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Then use the **merchantReference** to request the entire entity from the server.

```
terminal.retrieveSecureCard(merchantReference);
```

3. Give the proper treatment for each possible response:
    a) **onSecureCardResponse** callback: If the retrieving is successful, Your Application will receive the Secure Card entity related to the **merchantReference** informed (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.2 Secure Card Update** subsection). In this scenario your application should follow to the next step.
    b) **OnError** callback: If the registration is unsuccessful, You Application will receive the Error resulted from the retrieving (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection). In this scenario your application should give feedback to the User and restart the flow.
4. Show the information to Your Application's User.
    a) Card details: type; number; expiry date; card holder's name.
    b) Personal details: e-mail; mobile number; city; region; country; IP address.

    In this step you can usually add some business rules to make easier for the user to update the data avoiding submitting data that will return errors.

    An example: show and let the card's data be edited, unless you have a card reader device connected, in this case show the card's data, don't allow edition, and show a message informing that the card's data will be updated after the card reader device reads the card – that must be inserted or the updating will not occur.

    Depending on how you desire to control your application, you can apply many rules like the one described in the last paragraph.

    You can get the card's data that you need to show to the User in:

```
CoreSecureCardResponse.getCoreSecureCard();
```

5. Get the changes done by Your Application's User and change the information you need in the **CoreSecureCard** (**CoreSecureCardResponse**.coreSecureCard), before submitting again.

```
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);

// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

6. Adjust the **TokenMethod** – you need to consider that data received in **CoreSecureCardResponse**.getCoreSecureCard():

   a) If the card's data in entered in a Keyed way, create a **CoreKeyedSecureCard** and set the card's data, then set this **CoreKeyedSecureCard** to a new CoreTokenMethod. After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Data Method - Keyed
CoreKeyedSecureCard coreKeyedSecureCard = new CoreKeyedSecureCard();

// Card's Information
coreKeyedSecureCard.setCardType(cardType);
coreKeyedSecureCard.setCardNumber(cardNumber);
coreKeyedSecureCard.setExpiryDate(cardExpiryDate);
coreKeyedSecureCard.setCardHolderName(cardHolderName);
coreKeyedSecureCard.setCvv(cardCVV);

// Card's Token Method
CoreTokenMethod tokenMethod = new CoreTokenMethod(coreKeyedSecureCard);

// Method Information
coreSecureCard.setCoreTokenMethod(tokenMethod);
```

   b) If the card's data is captured by OCR, create a **CoreOCRSecureCard** and set the card's data, then set this **CoreOCRSecureCard** to a new CoreTokenMethod. After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Data Method - OCR
CoreOCRSecureCard coreOCRSecureCard = new CoreOCRSecureCard();

// Card's Information
coreOCRSecureCard.setCardType(cardType);
coreOCRSecureCard.setCardNumber(cardNumber);
coreOCRSecureCard.setExpiryDate(cardExpiryDate);
coreOCRSecureCard.setCardHolderName(cardHolderName);

// Card's Token Method
CoreTokenMethod tokenMethod = new CoreTokenMethod(coreOCRSecureCard);

// Method Information
coreSecureCard.setCoreTokenMethod(tokenMethod);
```

7. Call the Terminal's method to finalize the edition.

```
terminal.editSecureCard(coreSecureCard);
```

8. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card updated (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.2 Secure Card Update** subsection).
    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.2.2 Registration Using a Card Reader Device – MSR or EMV

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Then use the **merchantReference** to request the entire entity from the server.

```
terminal.retrieveSecureCard(merchantReference);
```

3. Give the proper treatment for each possible response:

a) **onSecureCardResponse** callback: If the retrieving is successful, Your Application will receive the Secure Card entity related to the **merchantReference** informed (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.2 Secure Card Update** subsection). In this scenario your application should follow to the next step.

b) **OnError** callback: If the registration is unsuccessful, You Application will receive the Error resulted from the retrieving (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the failure response** subsection). In this scenario your application should give feedback to the User and restart the flow.

4. Show the information to Your Application's User.
   a) Card details: type; number; expiry date; card holder's name.
   b) Personal details: e-mail; mobile number; city; region; country; IP address.

   In this step you can usually add some business rules to make easier for the user to update the data avoiding submitting data that will return errors.

   An example: show the card's data but don't allow changes, unless you don't have a card reader device connected, in this case show the card's data and allow edition, but show a message informing that there's no card reader device connected, so the User needs to change manually or connect a device to use a card for that.

   Depending on how you desire to control your application, you can apply many rules like the one described in the last paragraph.

   You can get the card's data that you need to show to the User in:

```
CoreSecureCardResponse.getCoreSecureCard();
```

5. Get the changes done by Your Application's User and change the information you need in the **CoreSecureCard** (**CoreSecureCardResponse**.coreSecureCard), before submitting again.

```
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);

// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

6. Call the Terminal's method to finalize the edition.

```
terminal.editSecureCard(coreSecureCard);
```

7. Give the proper treatment to each possible response:

    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the Successful Response – 2.1.5.1.2 Secure Card Update** subsection).

    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the Failure Response** subsection).

Note: Maybe you didn't notice, but until here we didn't talk about getting the Card's data. That's because, when trying to implement the MSR and EMV, you don't need to add any information regarding the Card's data (like the CoreTokenMethod you did for Keyed and OCR). Your Application just needs to pass the CoreSecureCard to the Terminal method and the SDK will realize that the data was not inserted on a Keyed or OCR way, and then the SDK will communicate with the Card Reader Device to get the Card's Data, but that is transparent to Your Application, so don't worry. Just make sure that there's a Card Reader Device connected before submitting the CoreSecureCard to the Terminal method, or you are going to receive an error message saying that you need to connect first, before any reading be done by the Device.

2.3 Secure Card Delete

The exclusion of a Secure Card is something simple and works the same way for all the cases. All you need do is use the **merchantReference** and the Secure Card's token of the **CoreSecureCard** you want deleted.

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Create a **CoreSecureCard**;

```
CoreSecureCard coreSecureCard = new CoreSecureCard();
```

3. Set the **merchantReference** from the Secure Card you select in the first step, to the **CoreSecureCard**.

```
// Merchant's Reference and token
coreSecureCard.setMerchantReference(merchantReference);
```

4. Call the Terminal's method to finalize the exclusion.

```
terminal.deleteSecureCard(coreSecureCard);
```

5. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card deleted (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.3 Secure Card Delete** subsection).
    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the Failure Response** subsection).

2.4 Secure Card Registration by Sale – Additional

Before we start talking about how to handle the responses, we need to see one more possibility for the Secure Card features: the registration of a Secure Card during a Sale.

In this scenario, it's reasonable to say that Your Application has almost all the essential information for a CoreSecureCard captured during the Sale transaction, so to make a Secure Card registration easier, you can use a **Sale** to register the card's data, already informed during the Sale, into a **SecureCard**.

Note: Be aware that this implementation in done inside an already existing "yourSaleMethod", just to take advantage of the fact that all the essential data for a Secure Card is already informed during a Sale.

The code sample below shows the code we are going to consider as **yourSaleMethod()** for our examples.

```
public void yourSaleMethod(){

 //Your sale method implementation

 //The Secure Card implementation
 /** All the steps shown in "Secure Card Registration by Sale", go in here, before the submitting
  */

 // Sale submitting
 terminal.processSale(sale);

}
```

As all the main information for a **CoreSecureCard** is already on a **Sale**, you just need to add a **CoreSecureCard** object, with just a **merchantReference** attribute, to the Sale.

In this kind of registration, Your Application doesn't need to differentiate between Keyed, OCR, MSR and EMV Methods, once the Sale already defined that when captured the card's data, so you just need to adjust **your*Sale*Method()**.

1. Create a **CoreSecureCard**

```
// Secure Card Object
CoreSecureCard coreSecureCard = new CoreSecureCard();
```

2. Set all the personal details to it, if you have them – the personal details are optional, so you just need to implement if you want or need to;

```
// Customer Information
coreSecureCard.setEmail(email);
coreSecureCard.setMobileNumber(mobilePhone);
coreSecureCard.setCity(city);
coreSecureCard.setRegion(region);
coreSecureCard.setCountry(country);
coreSecureCard.setIpAddress(ipAddress);
```

3. Set the **merchantReference** to the **CoreSecureCard**.

```
// Merchant's Reference -> Secure Card
coreSecureCard.setMerchantReference(merchantReference);
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

4. Associate **CoreCustomField** to your **SecureCard**.

```
// Custom Fields for Secure Card
List<CoreSecureCard.CoreCustomField> customFieldsList = new ArrayList();
customFieldsList.add(new CoreSecureCard.CoreCustomField("ReferenceID", valueRID));
customFieldsList.add(new CoreSecureCard.CoreCustomField("InvoiceNumber", valueIN));
coreSecureCard.setCustomFields(customFieldsList);
```

If you want, you can use Custom Fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the Terminal you are using to connect to the Gateway

5. Add the Secure Card created to the Sale – before the submitting the Sale using the Terminal.

```
// Secure Card -> Sale
sale.setCoreSecureCard(coreSecureCard);
```

6. Give the proper treatment to each possible response:

   a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.1.5 Secure Card Response – 2.1.5.1 Handling the successful response – 2.1.5.1.4 Secure Card Registration by Sale** subsection).

   b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.1.5 Secure Card Response – 2.1.5.2 Handling the Failure Response** subsection).

2.5 Secure Card Response

2.5.1 Handling the Successful Response

You need to consider that the responses are received in callback methods. That means Your Application needs to address properly, for each possible response, what to do. For instance: for the successful response

(**onSecureCardResponse** and **onSaleResponse**) implementation, Your Application will need to address all the service call responses that your application may receive.

The SDK is going to use **onSecureCardResponse(CoreSecureCardResponse response)**, method from **CoreAPIListener**, that your main class needs to implement, as the channel to send the successful responses to Your Application, regarding the **Secure Card Registration**, **Secure Card Update** and **Secure Card Delete**.

2.5.1.1 Secure Card Registration

Save the token, and if you want, and the description – the description can be used as a message to Your Application's User, if you prefer, but we recommend you create your own application's messages to provide "user friendly" feedback that fits your needs.

The description for this case will be "*Secure Card Saved*"!

```
@Override
public void onSecureCardResponse(CoreSecureCardResponse response) {
 // other implementation your application may require

 // Scenario Treatment 01 – Secure Card Registration
 String token = response.getSecureCardReference();
 String description = response.getDescription();

 // implementation to save the token registered – you may save or show the description too
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Registration of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.2 Secure Card Update

Consider that the update has actually two responses: the first one regarding the retrieve of the **CoreSecureCard**, by the **merchantReference**, and the receiving of the updated **CoreSecureCard**, after the changes.

In the first case, get the **CoreSecureCard** to be used by the update method, do the updating, then receive and treat the successful response.

Finally, save the token, and if you want, the description – the description can be used as a message to Your Application's User, if you prefer, but we recommend you create your own application's messages to provide "user friendly" feedback that fits your needs.

The description for this case will be "*Secure Card Updated*"!

```
@Override
public void onSecureCardResponse(CoreSecureCardResponse response) {
 // other implementation your application may require

 // Scenario Treatment 02 – Secure Card Update
 CoreSecureCard coreSecureCard = response.getCoreSecureCard();
 // Rest of the Secure Card update treatment

 String token = response.getSecureCardReference();
 String description = response.getDescription();

 // implementation to save the token registered – you may save or show the description too
 // implementation to compare the tokens, if you think it's necessary
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Update of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.3 Secure Card Delete

In this case the token returned by the **response.getSecureCardReference()** is *null*. We don't recommend that your application erase the references of the deleted Secure Card physically. The best option would be just disable the access to the Secure Card (just delete logically).

The description for this case will be "*Secure Card Deleted*"!

```
@Override
public void onSecureCardResponse(CoreSecureCardResponse response) {
 // other implementation your application may require

 // Scenario Treatment 03 – Secure Card Delete
 String token = response.getSecureCardReference();
 String description = response.getDescription();

 // implementation to save the token registered – you may save or show the description too
 // implementation to compare the tokens, if you think it's necessary
 // implementation to realize the logical exclusion of the token/ Secure Card
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Deleting of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.4 Secure Card Registration by Sale

In the case you used the option of **Secure Card Registration by Sale**, you need to adjust, your **onSaleResponse(final CoreSaleResponse response)** so you can get the token information.

In this scenario there's no description just the token represented by **response.cardReferenceNumber**, so you may as well just choose to add some information to your normal Sale feedback message, to let Your Application's User knows that, besides the Sale, the Secure Card Registration was successful too.

```
@Override
public void onSaleResponse(final CoreSaleResponse response) {
 //Your sale response implementation

 // If a Secure Card reference (token) was returned
 if(response.getCardReferenceNumber()!= null) {

   String token = response.getSecureCardReference();

   // implementation to save the token registered – you may save or show the description too
   // implementation to add Secure Card registration success info to the sale success feedback
 }

 // implementation to give feedback to the user
}
```

2.5.2 Handling the Failure Response

The **onError(CoreError coreError, String s),** method from **CoreAPIListener** interface – interface that your main class needs to implement – is going to be use by the SDK as the channel to send the failure responses to your application.

If you receive a **onError()** callback after executing any of the requests presented here, that means the request had some problem and you need to address that.

We recommend that your application log the error and the message returned, give a feedback to your application's user and show a "*try again*" message.

```
@Override
public void onError(CoreError coreError, String s) {
 // implementation to log the error returned
 // implementation to build the feedback message
 // implementation to give feedback to the user
}
```

Additionally to that, you can, if desired, implement some complementary treatment for each type of request that went wrong – for that you are going to need to know the request done first.

IOS Implementation

2.1 Secure Card Registration

In this implementation your application will have basically 3 (three) options for registering the Secure Card:

- **Registration without a card reader device – Keyed:** in this case, your application is going to ask to its User the card's details and, if it's necessary to your application, the owner's details.
- **Registration without a card reader device – OCR:** in this case, your application is going to provide the card's details, captured by a OCR solution, except the CVV, and if it's necessary to your application, the owner's details.

- **Registration using a card reader device – MSR or EMV:** in this case you application just worries about providing the card owner's details, if desired, once the SDK encapsulates all the logics involved in capturing the data from the card reader, being MSR or EMV.

2.1.1 Registration Without a Card Reader Device – Keyed

1. Capture the customer's data in your application – you decide the way.
    a) Card details: type; number; expiry date; card holder's name; CVV.
    b) Personal details: e-mail; mobile number; city; region; country; IP address.
2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard* coreSecureCard = [[CoreSecureCard alloc] init];

// Customer Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Create a **CoreTokenMethod** and set a **CoreKeyedSecureCard** containing the card's data.

```
//Card's Data Method – Keyed
coreSecureCard.tokenMethod = [[CoreTokenMethod alloc] init];
coreSecureCard.tokenMethod.keyedSecureCard = [[CoreKeyedSecureCard alloc] init];

//Card's Information
coreSecureCard.tokenMethod.keyedSecureCard.cardType = cardType;
coreSecureCard.tokenMethod.keyedSecureCard.cardNumber = cardNumber;
coreSecureCard.tokenMethod.keyedSecureCard.cardHolderName = cardHolderName;
coreSecureCard.tokenMethod.keyedSecureCard.expiryDate = cardExpiryDate;
coreSecureCard.tokenMethod.keyedSecureCard.cvv = cardCVV;
```

4. Set the *merchantReference* to the **CoreSecureCard**.

```
// Merchant's Reference
coreSecureCard.merchantReference = merchantReference;
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the

CoreSecureCard *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

5. Associate **CoreCustomField** to your **SecureCard**.

```
//Custom Fields
coreSecureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[coreSecureCard.customFields addObject:field];
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

6. Call the Terminal's method to finalize the registration.

```
[terminal registerSecureCard:coreSecureCard];
```

7. Give the proper treatment to each possible response:
   a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.1 Secure Card Registration** subsection).
   b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.1.2 Registration Without a Card Reader Device – OCR

1. Capture the customer's data in your application – you decide the way, although using the OCR implementation comes from the assumption that your application is going to use an OCR solution

for reading the Card Details.

a) Card details: type; number; expiry date; card holder's name.

b) Personal details: e-mail; mobile number; city; region; country; IP address.

2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard* coreSecureCard = [[CoreSecureCard alloc] init];

// Customer Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Create a **CoreTokenMethod** and set a **CoreOCRSecureCard** containing the card's data.

```
//Card's Data Method – OCR
coreSecureCard.tokenMethod = [[CoreTokenMethod alloc] init];
coreSecureCard.tokenMethod.ocrSecureCard = [[CoreOCRSecureCard alloc] init];

//Card's Information
coreSecureCard.tokenMethod.ocrSecureCard.cardType = cardType;
coreSecureCard.tokenMethod.ocrSecureCard.cardNumber = cardNumber;
coreSecureCard.tokenMethod.ocrSecureCard.cardHolderName = cardHolderName;
coreSecureCard.tokenMethod.ocrSecureCard.expiryDate = cardExpiryDate;
```

4. Set the **merchantReference** to the **CoreSecureCard**.

```
// Merchant's Reference
coreSecureCard.merchantReference = merchantReference;
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

5. Associate **CoreCustomField** to your **SecureCard**.

```
//Custom Fields
coreSecureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[coreSecureCard.customFields addObject:field];
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

6. Call the Terminal's method to finalize the registration.

```
[terminal registerSecureCard:coreSecureCard];
```

7. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.1 Secure Card Registration** subsection).
    b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.1.3 Registration Using a Card Reader Device – MSR and EMV

1. Capture the customer's data in your application – you decide the way – except for the card's data:
    a) Personal details: e-mail; mobile number; city; region; country; IP address.
2. Create a **CoreSecureCard** object and set all the personal details to it.

```
CoreSecureCard* coreSecureCard = [[CoreSecureCard alloc] init];

// Customer Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;
```

The personal details are optional, so you just need to implement if you want to, but you need to create the **CoreSecureCard** object anyway, once this object is the main component of the feature.

3. Set the *merchantReference* to the **CoreSecureCard**.

```
// Merchant's Reference
coreSecureCard.merchantReference = merchantReference;
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a response.

4. Associate **CoreCustomField** to your **SecureCard**.

```
//Custom Fields
coreSecureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[coreSecureCard.customFields addObject:field];
```

If you want, you can use custom fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the **Terminal** you are using to connect to the Gateway, during the Secure Card Registration.

5. Call the Terminal's method to finalize the registration.

```
[terminal registerSecureCard:coreSecureCard];
```

6. Give the proper treatment to each possible response:
    a) **onSecureCardResponse** callback: If the registration is successful, your application will

receive the Secure Card registered (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.1 Secure Card Registration** subsection).

b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

Note: Maybe you didn't notice, but until here we didn't talk about getting the Card's data. That's because, when trying to implement the **MSR** and **EMV**, you don't need to add any information regarding the Card's data (like the **CoreTokenMethod** you did for **Keyed** and **OCR**). Your Application just needs to pass the **CoreSecureCard** to the **Terminal** method and the SDK will realize that the data was not inserted on a **Keyed** or **OCR** way, and then the SDK will communicate with the Card Reader Device to get the Card's Data, but that is transparent to Your Application, so don't worry. Just make sure that there's a Card Reader Device connected before submitting the **CoreSecureCard** to the **Terminal** method, or you are going to receive an error message saying that you need to connect first, before any reading be done by the Device.

2.2 Secure Card Update

In this implementation your application will have basically 2 (two) options for updating the Secure Card:

- **Update without a card reader device – Keyed or OCR:** in this case, your application is going to retrieve a Secure Card, change any data informed and submit the data again for update, while there's no card device reader connected.

- **Registration using a card reader device – MSR or EMV:** in this case, your application is going to retrieve a Secure Card, change any data informed, except for the card's data, and submit the data, once the SDK encapsulates all the logics involved in capturing the data from the card reader, being MSR or EMV.

Note: Be aware of the main data you need to inform before updating the Secure Card. The CoreSecureCardResponse retrieved contains the whole structure in CoreSecureCardResponse.getCoreSecureCard(). You can use this same object to submit the changes and just adjust the data, depending on what Your Application's User changes.

2.2.1 Update Without a Card Reader Device – Keyed or OCR

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Then use the **merchantReference** to request the entire entity from the server.

```
[terminal retrieveSecureCard: merchantReference];
```

3. Give the proper treatment for each possible response:
    a) **onSecureCardResponse** callback: If the retrieving is successful, Your Application will receive the Secure Card entity related to the **merchantReference** informed (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.2 Secure Card Update** subsection). In this scenario your application should follow to the next step.
    b) **OnError** callback: If the registration is unsuccessful, You Application will receive the Error resulted from the retrieving (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection). In this scenario your application should give feedback to the User and restart the flow.
4. Show the information to Your Application's User.
    a) Card details: type; number; expiry date; card holder's name.
    b) Personal details: e-mail; mobile number; city; region; country; IP address.

    In this step you can usually add some business rules to make easier for the user to update the data avoiding submitting data that will return errors.

    An example: show and let the card's data be edited, unless you have a card reader device connected, in this case show the card's data, don't allow edition, and show a message informing that the card's data will be updated after the card reader device reads the card – that must be inserted or the updating will not occur.

    Depending on how you desire to control your application, you can apply many rules like the one described in the last paragraph.

    You can get the card's data that you need to show to the User in:

```
CoreSecureCard* secureCard = secureCardResponse.coreSecureCard;
```

5. Get the changes done by Your Application's User and change the information you need in the **CoreSecureCard** (**CoreSecureCardResponse**.coreSecureCard), before submitting again.

```
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;

//Custom Fields
coreSecureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[coreSecureCard.customFields addObject:field];
```

6. Adjust the **TokenMethod** – you need to consider that data received in **CoreSecureCardResponse**.getCoreSecureCard():

   a) If the card's data in entered in a Keyed way, create a **CoreKeyedSecureCard** and set the card's data, then set this **CoreKeyedSecureCard** to a new CoreTokenMethod. After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Data Method - Keyed
CoreSecureCard* coreSecureCard = [[CoreSecureCard alloc] init];

// Card's Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;

// Card's Token Method
coreSecureCard.tokenMethod = [[CoreTokenMethod alloc] init];
coreSecureCard.tokenMethod.ocrSecureCard = [[CoreOCRSecureCard alloc] init];
coreSecureCard.tokenMethod.keyedSecureCard = [[CoreKeyedSecureCard alloc] init];
```

   b) If the card's data is captured by OCR, create a **CoreOCRSecureCard** and set the card's data, then set this **CoreOCRSecureCard** to a new CoreTokenMethod. After that, set this **CoreTokenMethod** to the **CoreSecureCard.**

```
// Card's Data Method - OCR
CoreSecureCard* coreSecureCard = [[CoreSecureCard alloc] init];

// Card's Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;

// Card's Token Method
coreSecureCard.tokenMethod = [[CoreTokenMethod alloc] init];
coreSecureCard.tokenMethod.ocrSecureCard = [[CoreOCRSecureCard alloc] init];
```

7. Call the Terminal's method to finalize the edition.

```
[terminal editSecureCard:secureCard];
```

8. Give the proper treatment to each possible response:

   a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card updated (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.2 Secure Card Update** subsection).

   b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

Note: You need to consider the connected device before submitting anything to the Terminal. If there's a device connected, the SDK is going to consider that is necessary to capture card's data from the card reader device, and in case that happens, the Keyed/ OCR data informed is going to be overwritten by the data collected from the device.

2.2.2 Update Using a Card Reader Device – MSR and EMV

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Then use the **merchantReference** to request the entire entity from the server.

```
[terminal retrieveSecureCard: merchantReference];
```

3. Give the proper treatment for each possible response:

a) **onSecureCardResponse** callback: If the retrieving is successful, Your Application will receive the Secure Card entity related to the **merchantReference** informed (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.2 Secure Card Update** subsection). In this scenario your application should follow to the next step.

b) **OnError** callback: If the registration is unsuccessful, You Application will receive the Error resulted from the retrieving (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection). In this scenario your application should give feedback to the User and restart the flow.

4. Show the information to Your Application's User.
   a) Card details: type; number; expiry date; card holder's name.
   b) Personal details: e-mail; mobile number; city; region; country; IP address.

   In this step you can usually add some business rules to make easier for the user to update the data avoiding submitting data that will return errors.

   An example: show the card's data but don't allow changes, unless you don't have a card reader device connected, in this case show the card's data and allow edition, but show a message informing that there's no card reader device connected, so the User needs to change manually or connect a device to use a card for that.

   Depending on how you desire to control your application, you can apply many rules like the one described in the last paragraph.

   You can get the card's data that you need to show to the User in:

```
CoreSecureCard* secureCard = secureCardResponse.coreSecureCard;
```

5. Get the changes done by Your Application's User and change the information you need in the **CoreSecureCard** (**CoreSecureCardResponse**.coreSecureCard), before submitting again.

```
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;

//Custom Fields
coreSecureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[coreSecureCard.customFields addObject:field];
```

6. Call the Terminal's method to finalize the edition.

```
[terminal editSecureCard:secureCard];
```

7. Give the proper treatment to each possible response:

   a) **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card updated (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.2 Secure Card Update** subsection).

   b) **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

2.3 Secure Card Delete

The exclusion of a Secure Card is something simple and works the same way for all the cases. All you need do is use the **merchantReference** and the Secure Card's token of the **CoreSecureCard** you want deleted.

1. Select the Secure Card you want to update between the Secure Card references you have – remember the **merchantReference** field informed during the registration of the Secure Cards that you were suggested to keep in Your Application.
2. Create a **CoreSecureCard**;

```
CoreSecureCard* secureCard = [[CoreSecureCard alloc] init];
```

3. Set the **merchantReference** from the Secure Card you select in the first step, to the **CoreSecureCard**.

```
// Merchant's Reference and token
coreSecureCard.merchantReference = merchantReference;
```

4. Call the Terminal's method to finalize the exclusion.

```
[terminal deleteSecureCard:secureCard];
```

5. Give the proper treatment to each possible response:
    a. **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card deleted (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.3 Secure Card Delete** subsection).
    b. **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

2.4 Secure Card Registration by Sale – Additional

Before we start talking about how to handle the responses, we need to see one more possibility for the Secure Card features: the registration of a Secure Card during a Sale.

In this scenario, it's reasonable to say that Your Application has almost all the essential information for a CoreSecureCard captured during the Sale transaction, so to make a Secure Card registration easier, you can use a **Sale** to register the card's data, already informed during the Sale, into a **SecureCard**.

Note: Be aware that this implementation in done inside an already existing "**yourSaleMethod**", just to take advantage of the fact that all the essential data for a Secure Card is already informed during a Sale.

The code sample below shows the code we are going to consider as **yourSaleMethod()** for our examples.

```
- (void) yourSaleMethod {
 //Your sale method implementation

 //The Secure Card implementation ------------------------------
 /* All the steps shown in "Secure Card Registration by Sale", go in here, before the submitting
  */

 // Sale submitting
 [terminal processSale:sale];
}
```

As all the main information for a **CoreSecureCard** is already on a **Sale**, you just need to add a **CoreSecureCard** object, with just a **merchantReference** attribute, to the Sale.

In this kind of registration, Your Application doesn't need to differentiate between Keyed, OCR, MSR and EMV Methods, once the Sale already defined that when captured the card's data, so you just need to adjust **your*Sale*Method()**.

1. Create a **CoreSecureCard**

```
//The Secure Card implementation ------------------------------
CoreSecureCard* secureCard = [[CoreSecureCard alloc] init];
```

2. Set all the personal details to it, if you have them – the personal details are optional, so you just need to implement if you want or need to;

```
// Customer Information
coreSecureCard.email = email;
coreSecureCard.mobileNumber = mobilePhone;
coreSecureCard.city = city;
coreSecureCard.region = region;
coreSecureCard.country = country;
coreSecureCard.ipAddress = ipAddress;
```

3. Set the **merchantReference** to the **CoreSecureCard**.

```
// Merchant's Reference -> Secure Card
secureCard.merchantReference = merchantReference;
```

This information regards the reference you use to distinguish the secure cards in your own way. You should decide how to generate this information, but be aware that this information is going to be required in all Secure Card operations – we suggest that you store this information associated to the **CoreSecureCard** *token* (**CoreSecureCardResponse.***secureCardReference*) that you may receive as a

response.

4. Associate **CoreCustomField** to your **SecureCard**.

```
// Custom Fields for Secure Card
secureCard.customFields = [[NSMutableArray alloc] init];
CoreCustomField *field = [[CoreCustomField alloc] init];

field.name = @"referenceID";
field.value = valueRID;

[secureCard.customFields addObject:field];
```

If you want, you can use Custom Fields to store any additional property of your Secure Cards. For that you need to know the **CustomField** list that you have set at the Terminal you are using to connect to the Gateway

5. Add the Secure Card created to the Sale – before the submitting the Sale using the Terminal.

```
// Secure Card -> Sale
sale.coreSecureCard = coreSecureCard;
```

6. Give the proper treatment to each possible response:
    a. **onSecureCardResponse** callback: If the registration is successful, your application will receive the Secure Card registered (See the **2.2.5 Secure Card Response – 2.2.5.1 Handling the successful response – 2.2.5.1.4 Secure Card Registration by Sale** subsection).
    b. **onError** callback: If the registration is unsuccessful, your application will receive the Error resulted from the request (See the **2.2.5 Secure Card Response – 2.2.5.2 Handling the Failure Response** subsection).

2.5 Secure Card Response

2.5.1 Handling the Successful Response

You need to consider that the responses are received in callback methods. That means Your Application needs to address properly, for each possible response, what to do. For instance: for the successful response (**onSecureCardResponse** and **onSaleResponse**) implementation, Your Application will need to address all the service call responses that your application may receive.

The SDK is going to use **onSecureCardResponse(CoreSecureCardResponse response)**, method from **CoreAPIListener**, that your main class needs to implement, as the channel to send the successful responses to Your Application, regarding the **Secure Card Registration**, **Secure Card Update** and **Secure Card Delete**.

2.5.1.1 Secure Card Registration

Save the token, and if you want, and the description – the description can be used as a message to Your Application's User, if you prefer, but we recommend you create your own application's messages to provide "user friendly" feedback that fits your needs.

The description for this case will be "*Secure Card Saved*"!

```
- (void) onSecureCardResponse:(CoreSecureCardResponse*)secureCardResponse{
 // other implementation your application may require

 // Scenario Treatment 01 – Secure Card Registration
 self.scToken = secureCardResponse.secureCardReference;
 self.scDescription = secureCardResponse.secureCardReference;

 // implementation to save the token registered – you may save or show the description too
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Registration of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.2 Secure Card Update

Consider that the update has actually two responses: the first one regarding the retrieve of the **CoreSecureCard**, by the **merchantReference**, and the receiving of the updated **CoreSecureCard**, after the changes.

In the first case, get the **CoreSecureCard** to be used by the update method, do the updating, then receive and treat the successful response.

Finally, save the token, and if you want, the description – the description can be used as a message to Your Application's User, if you prefer, but we recommend you create your own application's messages to provide "user friendly" feedback that fits your needs.

The description for this case will be "*Secure Card Updated*"!

```
- (void) onSecureCardResponse:(CoreSecureCardResponse*)secureCardResponse{
 // other implementation your application may require

 // Scenario Treatment 02 – Secure Card Update
 CoreSecureCard coreSecureCard = response.getCoreSecureCard();

 // Rest of the Secure Card update treatment

 self.scToken = secureCardResponse.secureCardReference;
 self.scDescription = secureCardResponse.secureCardReference;

 // implementation to save the token registered – you may save or show the description too
 // implementation to compare the tokens, if you think it's necessary
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Update of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.3 Secure Card Delete

In this case the token returned by the **response.getSecureCardReference()** is *null*. We don't recommend that your application erase the references of the deleted Secure Card physically. The best option would be just disable the access to the Secure Card (just delete logically).

The description for this case will be "*Secure Card Deleted*"!

```
- (void) onSecureCardResponse:(CoreSecureCardResponse*)secureCardResponse{
 // other implementation your application may require

 // Scenario Treatment 03 – Secure Card Delete
 self.scToken = secureCardResponse.secureCardReference;
 self.scDescription = secureCardResponse.secureCardReference;

 // implementation to save the token registered – you may save or show the description too
 // implementation to compare the tokens, if you think it's necessary
 // implementation to realize the logical exclusion of the token/ Secure Card
 // implementation to give feedback to the user
}
```

The code above is expected to be added to Your Application main controller that implements the **CoreAPIListener** interface from the SDK.

We recommend that Your Application use a control variable to know, at least, what was the request (in this case, Deleting of Secure Card), and separate the treatment code for this scenario – that's a recommendation for all the scenarios, actually. But that is not mandatory – just a hint to facilitate the flow control.

2.5.1.4 Secure Card Registration by Sale

In the case you used the option of **Secure Card Registration by Sale**, you need to adjust, your **onSaleResponse(final CoreSaleResponse response)** so you can get the token information.

In this scenario there's no description just the token represented by **response.cardReferenceNumber**, so you may as well just choose to add some information to your normal Sale feedback message, to let Your Application's User knows that, besides the Sale, the Secure Card Registration was successful too.

```
- (void) onSaleResponse:(CoreSaleResponse*)cardRequest{
 //Your sale response implementation

 // If a Secure Card reference (token) was returned
 if(cardRequest.secureCardReference != nil && cardRequest.secureCardReference.length>0) {

    self.scToken = secureCardResponse.secureCardReference;

    // implementation to save the token registered – you may save or show the description too
    // implementation to add Secure Card registration success info to the sale success feedback
 }

 // implementation to give feedback to the user
}
```

2.5.2 Handling the Failure Response

The **onError(CoreError coreError, String s)**, method from **CoreAPIListener** interface – interface that your main class needs to implement – is going to be use by the SDK as the channel to send the failure responses to your application.

If you receive a **onError()** callback after executing any of the requests presented here, that means the request had some problem and you need to address that.

We recommend that your application log the error and the message returned, give a feedback to your application's user and show a "*try again*" message.

```
- (void) onError:(CoreError)error withDescription:(NSString *)message {
 // implementation to log the error returned
 // implementation to build the feedback message
 // implementation to give feedback to the user
}
```

Additionally to that, you can, if desired, implement some complementary treatment for each type of request that went wrong – for that you are going to need to know the request done first.

3 Appendix A – Mobile SDK Code Samples

Android Samples

3.1 CoreAPIListener

```
/**
 * This class is used to listen for events which are coming back to the integrator.
 * When the integrator makes a request, response is processed here and sent back to him.
 * Integrator must implement this interface to listen for the responses.
 */
public interface CoreAPIListener {
  /**
   * Contains message coming from the SDK while processing the transaction.
   *
   * @param message enum value of a message,
   *            {@link CoreMessage}
   */
  void onMessage(CoreMessage message);
  /**
   * Fires after transaction has been processed successfully in response to processSale method. Contains transaction response object from the
server.
   *
   * @param response contains CoreSaleResponse object retrieved from the server,
   *            {@link CoreSaleResponse}
   */
  void onSaleResponse(CoreSaleResponse response);
  /**
   * This method is called after refund has been processed in response to processRefund method.
   *
   * @param response contains CoreRefundResponse object retrieved from the server,
   *            {@link CoreRefundResponse}
   */
  void onRefundResponse(CoreRefundResponse response);
  /**
   * Returns CoreTransactions object which consists of the list with the last 10 transactions in response to getTransactions method.
   *
   * @param response contains CoreTransactions object retrieved from the server,
   *            {@link CoreTransactions}
   */
  void onTransactionListResponse(CoreTransactions response);
  /**
   * Returns URL which needs to be used to sign in into SelfCare System in response to requestSecuredUrl method.
   *
   * @param url holds secured URL, allowed object is
   *        {@link String}
   */
  void onLoginUrlRetrieved(String url);
  /**
   * Fires when signature is required for the transaction to be processed.
   *
   * @param signature holds the signature, allowed object is
   *            {@link CoreSignature}
   */
  void onSignatureRequired(CoreSignature signature);
  /**
   * This method is triggered after error has occurred in the SDK.
   *
   * @param error   allowed object is
```

```
 *            {@link CoreError}
 * @param message description of an error
 */
void onError(CoreError error, String message);
/**
 *This method is triggered when there is an error coming from the device.
 *
 * @param error   allowed object is
 *            {@link CoreError}
 * @param message description of an error
 */
void onDeviceError(CoreDeviceError error, String message);
/**
 * Sends back terminal settings object in response to init method.
 *
 * @param settings corresponds to terminal settings from the server,
 *             {@link CoreSettings}
 */
void onSettingsRetrieved(CoreSettings settings);
/**
 * Fires when device gets connected.
 *
 * @param type      represents connected device ,
 *              {@link DeviceEnum}
 * @param deviceInfo device information
 */
void onDeviceConnected(DeviceEnum type, HashMap<String, String> deviceInfo);
/**
 * Fires when device gets disconnected.
 *
 * @param type represents disconnected device ,
 *          {@link DeviceEnum}
 */
void onDeviceDisconnected(DeviceEnum type);
/**
 *This method is called when the application selection is required.
 *
 * @param applications to choose from
 */
void onSelectApplication(ArrayList<String> applications);
/**
 *This method is called when the bluetooth device selection is required.
 *
 * @param devices to choose from
 */
void onSelectBTDevice(ArrayList<String> devices);
/**
 *This method is called when there is a problem connecting to a device.
 */
void onDeviceConnectionError();
/**
 *This method is called when auto-config progress updates.
 */
void onAutoConfigProgressUpdate(String progress);
/**
 *This method is called after reversal has been processed.
 *
 * @param reversalResponse - contains reversal response from the server
 */
void onReversalRetrieved(CoreResponse reversalResponse);
/**
 * Fires when the serial post needs to be selected.
 *
 * @param ports to choose from
 */
```

```
  void onSelectSerialPort(ArrayList<String> ports);
  /**
   * Information about the device.
   *
   * @param deviceInfo - contains device information
   */
  void onDeviceInfoReturned(HashMap<String, String> deviceInfo);
  /**
   * Fires after the secure card has been registered/edited/deleted successfully.
   * @param secureCardResponse contains CoreSecureCardResponse object retrieved from the server
   */
  void onSecureCardResponse(CoreSecureCardResponse secureCardResponse);
}
```

3.2 CoreKeyedSecureCard

```
public class CoreKeyedSecureCard extends CoreOCRSecureCard {
   private String cvv;
   public String getCvv() {
      return cvv;
   }
   public void setCvv(String cvv) {
      this.cvv = cvv;
   }
}
```

3.3 CoreTokenMethod

```
public class CoreTokenMethod {
   private CoreKeyedSecureCard keyedSecureCard;
   private CoreEmvSecureCard emvSecureCard;
   private CoreTrackSecureCard trackSecureCard;
   private CoreOCRSecureCard ocrSecureCard;
   public CoreTokenMethod() {
   }
   public CoreTokenMethod(CoreKeyedSecureCard coreKeyedPaymentMethod) {
      this.keyedSecureCard = coreKeyedPaymentMethod;
   }
   public CoreTokenMethod(CoreEmvSecureCard coreEmvPaymentMethod) {
      this.emvSecureCard = coreEmvPaymentMethod;
   }
   public CoreTokenMethod(CoreTrackSecureCard coreTrackSecureCard) {
      this.trackSecureCard = coreTrackSecureCard;
   }
   public CoreKeyedSecureCard getKeyedSecureCard() {
      return keyedSecureCard;
   }
   public void setKeyedSecureCard(CoreKeyedSecureCard keyedSecureCard) {
      this.keyedSecureCard = keyedSecureCard;
   }
   public CoreEmvSecureCard getEmvSecureCard() {
      return emvSecureCard;
   }
   public void setEmvSecureCard(CoreEmvSecureCard emvSecureCard) {
      this.emvSecureCard = emvSecureCard;
```

```
  }
  public CoreTrackSecureCard getTrackSecureCard() {
    return trackSecureCard;
  }
  public void setTrackSecureCard(CoreTrackSecureCard trackSecureCard) {
    this.trackSecureCard = trackSecureCard;
  }
  public CoreOCRSecureCard getOcrSecureCard() {
    return ocrSecureCard;
  }
  public void setOcrSecureCard(CoreOCRSecureCard ocrSecureCard) {
    this.ocrSecureCard = ocrSecureCard;
  }
}
```

3.4 CoreResponse

```
/**
 * This class represents transaction response object.
 */
public class CoreResponse extends ResponseJson implements Serializable{
  /**
   * The {@link String} instance representing card holder name.
   */
  private String cardHolderName;
  /**
   * The {@link String} instance representing card number.
   */
  private String cardNumber;
  /**
   * The {@link String} instance representing expiry date.
   */
  private String expiryDate;
  /**
   * The {@link AvsResponseCode} instance representing response code.
   */
  private AvsResponseCode avsResponseCode;
  /**
   * The {@link String} instance representing approval code.
   */
  private String approvalCode;
  /**
   * The {@link String} instance representing authorized amount.
   */
  private BigDecimal authorizedAmount;
  /**
   * The {@link CvvResponseCode} instance representing cvv response code.
   */
  private CvvResponseCode cvvResponseCode;
  /**
   * The {@link String} instance representing currency.
   */
  private String currency;
  /**
   * The {@link String} instance representing unique reference.
   */
  private String uniqueRef;
  /**
   * The {@link String} instance representing date and time.
   */
```

```
private String dateTime;
/**
 *The {@link String} instance representing description.
 */
private String description;
/**
 *The {@link String} instance representing code.
 */
private String code;
private String cardType;
private String cardReferenceNumber;
public CoreResponse() {
}
/**
 *This AvsResponseCode represents AVS response code.
 *
 * @return aVSResponseCode
 */
public AvsResponseCode getAVSResponseCode() {
  return avsResponseCode;
}
/**
 *This String represents approval code for each transaction.
 *
 * @return approvalCode
 */
public String getApprovalCode() {
  return approvalCode;
}
/**
 *This BigDecimal represents amount set for each transaction.
 *
 * @return authorizedAmount
 */
public BigDecimal getAuthorizedAmount() {
  return authorizedAmount;
}
/**
 *This CvvResponseCode represents cvv response code 3 digit number found on the back of credit/debit card.
 *
 * @return cVVResponseCode
 */
public CvvResponseCode getCVVResponseCode() {
  return cvvResponseCode;
}
/**
 *This String represents currency for example: Dollar,Euro.
 *
 * @return currency
 */
public String getCurrency() {
  return currency;
}
/**
 *This String represents unique reference assigned to each transaction.
 *
 * @return uniqueRef
 */
public String getUniqueRef() {
  return uniqueRef;
}
/**
 *This String represents the date and time when transaction has been made.
 *
 * @return dateTime
```

```
 */
public String getDateTime() {
  return dateTime;
}
/**
 * This String represents description if transaction has been approved or not declined.
 *
 * @return description
 */
public String getDescription() {
  return description;
}
/**
 * This String represents code if transaction has been approved or not approved.
 *
 * @return code
 */
public String getCode() {
  return code;
}
/**
 * This String represents card type.
 *
 * @return cardType
 */
public String getCardType() {
  return cardType;
}
/**
 * Sets the value of cardType property.
 */
public void setCardType(String cardType) {
  this.cardType = cardType;
}
/**
 * Sets the value of avsResponseCode property.
 */
public void setAvsResponseCode(AvsResponseCode avsResponseCode) {
  this.avsResponseCode = avsResponseCode;
}
/**
 * Sets the value of approvalCode property.
 */
public void setApprovalCode(String approvalCode) {
  this.approvalCode = approvalCode;
}
/**
 * Sets the value of authorizedAmount property.
 */
public void setAuthorizedAmount(BigDecimal authorizedAmount) {
  this.authorizedAmount = authorizedAmount;
}
/**
 * Sets the value of cvvResponseCode property.
 */
public void setCvvResponseCode(CvvResponseCode cvvResponseCode) {
  this.cvvResponseCode = cvvResponseCode;
}
/**
 * Sets the value of currency property.
 */
public void setCurrency(String currency) {
  this.currency = currency;
}
/**
```

```
 * Sets the value of uniqueRef property.
 */
public void setUniqueRef(String uniqueRef) {
  this.uniqueRef = uniqueRef;
}
/**
 * Sets the value of dateTime property.
 */
public void setDateTime(String dateTime) {
  this.dateTime = dateTime;
}
/**
 * Sets the value of description property.
 */
public void setDescription(String description) {
  this.description = description;
}
/**
 * Sets the value of code property.
 */
public void setCode(String code) {
  this.code = code;
}
/**
 * This String represents card holder name.
 *
 * @return cardType
 */
public String getCardHolderName() {
  return cardHolderName;
}
/**
 * Sets the value of cardHolderName property.
 */
public void setCardHolderName(String cardHolderName) {
  this.cardHolderName = cardHolderName;
}
/**
 * This String represents card number.
 *
 * @return cardType
 */
public String getCardNumber() {
  return cardNumber;
}
/**
 * Sets the value of cardNumber property.
 */
public void setCardNumber(String cardNumber) {
  this.cardNumber = cardNumber;
}
/**
 * This String represents expiry date of the card.
 *
 * @return cardType
 */
public String getExpiryDate() {
  return expiryDate;
}
/**
 * Sets the value of expiryDate property.
 */
public void setExpiryDate(String expiryDate) {
  this.expiryDate = expiryDate;
}
```

```
  public String getCardReferenceNumber() {
    return cardReferenceNumber;
  }
  public void setCardReferenceNumber(String cardReferenceNumber) {
    this.cardReferenceNumber = cardReferenceNumber;
  }
}
```

3.5 CoreSaleResponse

```
/**
 * This class represents sale response object
 */
public class CoreSaleResponse extends CoreResponse implements Serializable {
  /**
   * The InputMethod instance representing transaction type.
   */
  private InputMethod inputMethod;
  private ArrayList<CoreEmvTag> emvTags = new ArrayList<>();
  /**
   * CoreSaleResponse Constructor sets cardHolderName,cardNumber,cardType and expiryDate based on the response from the server.
   */
  public CoreSaleResponse() {
  }
  /**
   * The InputMethod instance representing input method.
   *
   * @return inputMethod
   */
  public InputMethod getInputMethod() {
    return inputMethod;
  }
  /**
   * The String instance representing emvTags.
   *
   * @return emvTags
   */
  public ArrayList<CoreEmvTag> getEmvTagListArray() {
    return emvTags;
  }
  public void setInputMethod(InputMethod inputMethod) {
    this.inputMethod = inputMethod;
  }
  /**
   * Gets EMV tags.
   *
   * @return emvTags
   */
  public ArrayList<CoreEmvTag> getEmvTags() {
    return emvTags;
  }
  public void setEmvTags(ArrayList<CoreEmvTag> emvTags) {
    this.emvTags = emvTags;
  }
}
```

3.6 CoreSecureCard

```
public class CoreSecureCard {
  private String merchantReference;
  private String cardReference;
  @Optional private String deviceType;
  private CoreTokenMethod coreTokenMethod;
  @Optional private String email;
  @Optional private String mobileNumber;
  @Optional private String city;
  @Optional private String region;
  @Optional private String country;
  @Optional private String ipAddress;
  @Optional private List<CoreCustomField> customFields;
  @Optional private CoreCardEntryMode coreCardEntryMode;
  public CoreSecureCard() {
  }
  public CoreSecureCard(String merchantReference) {
    this.merchantReference = merchantReference;
  }
  public String getMerchantReference() {
    return merchantReference;
  }
  public void setMerchantReference(String merchantReference) {
    if(merchantReference != null && !merchantReference.isEmpty())
    this.merchantReference = merchantReference;
  }
  public String getCardReference() {
    return cardReference;
  }
  public void setCardReference(String cardReference) {
    this.cardReference = cardReference;
  }
  public String getEmail() {
    return email;
  }
  public void setEmail(String email) {
    this.email = email;
  }
  public String getMobileNumber() {
    return mobileNumber;
  }
  public void setMobileNumber(String mobileNumber) {
    this.mobileNumber = mobileNumber;
  }
  public String getCity() {
    return city;
  }
  public void setCity(String city) {
    this.city = city;
  }
  public String getRegion() {
    return region;
  }
  public void setRegion(String region) {
    this.region = region;
  }
  public String getCountry() {
    return country;
  }
  public void setCountry(String country) {
    this.country = country;
  }
  public String getIpAddress() {
```

```
    return ipAddress;
  }
  public void setIpAddress(String ipAddress) {
    this.ipAddress = ipAddress;
  }
  public List<CoreCustomField> getCustomFields() {
    return customFields;
  }
  public void setCustomFields(List<CoreCustomField> customFields) {
    this.customFields = customFields;
  }
  public CoreTokenMethod getCoreTokenMethod() {
    return coreTokenMethod;
  }
  public void setCoreTokenMethod(CoreTokenMethod coreTokenMethod) {
    this.coreTokenMethod = coreTokenMethod;
  }
  public String getDeviceType() {
    return deviceType;
  }
  public CoreCardEntryMode getCoreCardEntryMode() {
    return coreCardEntryMode;
  }
  public void setCoreCardEntryMode(CoreCardEntryMode coreCardEntryMode) {
    this.coreCardEntryMode = coreCardEntryMode;
  }
  public void setDeviceType(String deviceType) {
    this.deviceType = deviceType;
  }
  /**
   * Inner class which represents the custom fields
   */
  /**
   * Inner class which represents the custom fields
   */
  public static class CoreCustomField{
    private String name;
    private String value;
    public CoreCustomField(){
    }
    public CoreCustomField(String name, String value){
      this.name = name;
      this.value = value;
    }
    public String getName() {
      return name;
    }
    public void setName(String name) {
      this.name = name;
    }
    public String getValue() {
      return value;
    }
    public void setValue(String value) {
      this.value = value;
    }
  }
}
```

3.7 CoreSecureCardResponse

```
public class CoreSecureCardResponse {
  private String secureCardReference;
  private String secureCardDescription;
  private CoreSecureCard coreSecureCard;
  public String getSecureCardReference() {
    return secureCardReference;
  }
  public void setSecureCardReference(String secureCardReference) {
    this.secureCardReference = secureCardReference;
  }
  public String getSecureCardDescription() {
    return secureCardDescription;
  }
  public void setSecureCardDescription(String secureCardDescription) {
    this.secureCardDescription = secureCardDescription;
  }
  public CoreSecureCard getCoreSecureCard() {
    return coreSecureCard;
  }
  public void setCoreSecureCard(CoreSecureCard coreSecureCard) {
    this.coreSecureCard = coreSecureCard;
  }
}
```

IOS Samples

3.1 CoreAPIListener

```
/*!
 * @brief This class is used to listen for events which are coming back to the integrator.
 * When the integrator makes a request, response is processed here and sent back to him.
 * integrator must implement this delegate to listen for the responses.
 */
@protocol CoreAPIListener <NSObject>

/*!
 * This method is triggered after error has occurred in the SDK.
 * @param error contains typedef enum of an error.
 */
-(void)onError:(CoreError) error withDescription:(NSString*) message;

/*!
 * This method is triggered when there is an error coming from the device.
 * @param deviceError contains typedef enum of an error.
 */
-(void)onDeviceError:(CoreDeviceError) deviceError withDescription:(NSString*) message;

/*!
 * Fires when signature is required for the transaction to be processed.
 * @param signature holds the signature.
 */
-(void)onSignatureRequired:(CoreSignature*)signature;

/*!
 * Returns URL which needs to be used to sign in into SelfCare System in response to requestSecuredUrl method.
 * @param url
 */
-(void)onLoginUrlRetrieved:(NSString *)url;

/*!
 * Contains message coming from the SDK while processing the transaction.
 * @param message enum value of a message
 */
-(void)onMessage:(CoreMessage)message;

/*!
 * This method is called after refund has been processed in response to processRefund method.
 * @param refund contains CoreRefundResponse object retrieved from the server
 */
-(void)onRefundResponse:(CoreRefundResponse*)refund;

/*!
 * Fires after transaction has been processed successfully in response to processSale method. Contains transaction response object from the
server.
 * @param sale contains CoreSaleResponse object retrieved from the server
 */
-(void)onSaleResponse:(CoreSaleResponse*)sale;

/*!
 * Fires after the secure card has been registered/edited/deleted successfully.
 * @param secureCardResponse contains CoreSecureCardResponse object retrieved from the server
 */
-(void)onSecureCardResponse:(CoreSecureCardResponse*)secureCardResponse;

/*!
```

```
 * Returns CoreTransactions object which consists of the list with the last 10 transactions in response to getTransactions method.
 * @param transaction contains CoreTransactions object retrieved from the server
 */
-(void)onTransactionListResponse:(CoreTransactions*)transaction;

/*!
 * Sends back terminal settings object in response to init method.
 * @param settings holds the settings
 */
-(void)onSettingsRetrieved:(CoreSettings*)settings;

/*!
 * Fires when device gets connected.
 * @param type represents connected device
 */
-(void)onDeviceConnected:(DeviceEnum)type withDeviceInfo:(NSDictionary*) deviceInfo;

/*!
 * Fires when device gets disconnected.
 * @param type represents disconnected device
 */
-(void)onDeviceDisconnected:(DeviceEnum)type;

/*!
 * This method is called when the application selection is required.
 * @param applications represents the array of applications to choose from
 */
-(void)onSelectApplication:(NSArray*)applications;

/*!
 * This method is called when the bluetooth device selection is required.
 * @param type represents the array of devices to choose from
 */
-(void)onSelectBTDevice:(NSArray*)devices;

/*!
 * This method is called when there is a problem connecting to a device.
 */
-(void) onDeviceConnectionError;
/*!
 * This method is called after reversal has been processed.
 * @param sale contains reversal object retrieved from the server
 */
-(void) onReversalRetrieved:(CoreResponse*)reversalResponse;
/*!
 * Information about the device.
 * @param deviceInfo returned
 */
- (void) onDeviceInfoReturned:(NSDictionary*)deviceInfo;

@end
```

3.2 CoreKeyedSecureCard

```
/*!
 * @brief CoreKeyedMethod object used for card keyed data.
 *
 * <code>
 *          CoreKeyedMethod keyedMethod = [[CoreKeyedMethod alloc] init];
```

```
*           keyedMethod.cardType = @"VISA";
*           keyedMethod.cardNumber = @"123456789123";
*           keyedMethod.cardHolderName = @"Card Holder Name";
*           keyedMethod.expiryDate = @"1218";
*           keyedMethod.cvv = @"123";
* </code>
*
*/
@interface CoreKeyedSecureCard : CoreOCRSecureCard
/*! @brief This property represents the card cvv. */
@property NSString* cvv;

@end
```

3.3 CoreResponse

```
/*!
 * @brief This class represents the CoreResponse object.
 */
@interface CoreResponse : NSObject
/*! @brief This property represents approval Code. */
@property NSString* approvalCode;
/*! @brief This property represents amount. */
@property NSString* authorizedAmount;
/*! @brief This property represents cvv response code. */
@property NSString* cvvResponseCode;
/*! @brief This property represents currency. */
@property Currency currency;
/*! @brief This property represents unique reference. */
@property NSString* uniqueRef;
/*! @brief This property represents AVS response code. */
@property NSString* avsResponseCode;
/*! @brief This property represents date and time. */
@property NSString* dateTime;
/*! @brief This property represents description code. */
@property NSString* descriptionCode;
/*! @brief This property represents code. */
@property NSString* code;
/*! @brief This property represents card holder name. */
@property NSString* cardHolderName;
/*! @brief This property represents card number. */
@property NSString* cardNumber;
/*! @brief This property represents card type. */
@property NSString* cardType;
/*! @brief This property represents expiry date. */
@property NSString* expiryDate;
/*! @brief This property indicates if a reversal occurred or not. */
@property BOOL didPerformReversal;
/*! @brief This property represents the reference for the secure card. */
@property NSString *secureCardReference;

@end
```

3.4 CoreSaleResponse

```
/*!
 * @brief This class represents the sale response object
 */
```

```
@interface CoreSaleResponse : CoreResponse
/*! @brief This property represents transaction type. */
@property CoreTransactionInputMethod inputMethod;
/*! @brief This property represents aid. */
@property NSMutableArray *emvTagListArray;
/*! @brief Converts object to NSDictionary. */
-(NSMutableDictionary*)getAsJsonObject:(CoreSaleResponse *) response;

@end

@end
```

3.5 CoreSecureCard

```
/*!
 * @brief This class represents the card to be registered/edited/deleted as a secure card.
 *
 * <code>
 *          CoreSecureCard secureCard = [[CoreSecureCard alloc] init];
 *          secureCard.merchantReference = @"MR01";
 *          ...
 * </code>
 *
 */
@interface CoreSecureCard : NSObject
/*! @brief This property represents the reference for the new secure card. */
@property NSString* merchantReference;
/*! @brief This property represents the reference of the secure card previously saved. */
@property NSString* cardReference;
/*! @brief This property represents the mode the card data was entered. */
@property CoreCardEntryMode cardEntryMode;
/*! @brief This property represents the device id. */
@property NSString* deviceId;
/*! @brief This property represents the device is being used */
@property NSString* deviceType;
/*! @brief This property represents the operator name. */
@property NSString* operatorName;
/*! @brief This property represents the card holder email. */
@property NSString* email;
/*! @brief This property represents the card holder mobile number. */
@property NSString* mobileNumber;
/*! @brief This property represents the card holder city. */
@property NSString* city;
/*! @brief This property represents the card holder region. */
@property NSString* region;
/*! @brief This property represents the card holder country. */
@property NSString* country;
/*! @brief This property represents the ip address. */
@property NSString* ipAddress;
/*! @brief This property represents the list of custom fields. */
@property NSMutableArray* customFields;
/*! @brief This property represents the method used to retrieve the card data. */
@property CoreTokenMethod* tokenMethod;

@end
```

3.6 CoreTokenPaymentMethod

```
/*!
 * @brief CoreTokenMethod object used for secure card register/edit.
 *
 * <code>
 *            CoreTokenMethod tokenMethod = [[CoreTokenMethod alloc] init];
 *            tokenMethod.keyedSecureCard = [[CoreKeyedSecureCard alloc] init];
 *            tokenMethod.trackSecureCard = [[CoreTrackSecureCard alloc] init];
 *            tokenMethod.emvSecureCard = [[CoreEmvSecureCard alloc] init];
 *  tokenMethod.ocrSecureCard = [[CoreOCRSecureCard alloc] init];
 * </code>
 *
 */
@interface CoreTokenMethod : NSObject
/*! @brief This property represents the keyed payment method. */
@property CoreKeyedSecureCard *keyedSecureCard;
/*! @brief This property represents the track payment method. */
@property CoreTrackSecureCard* trackSecureCard;
/*! @brief This property represents the EMV payment method. */
@property CoreEmvSecureCard* emvSecureCard;
/*! @brief This property represents the OCR payment method. */
@property CoreOCRSecureCard* ocrSecureCard;

@end
```

3.7 CoreSecureCardResponse

```
/*!
 * @brief This class represents the secure card response object
 */
@interface CoreSecureCardResponse : CoreResponse
/*! @brief This property represents the description for the secure card operation. */
@property NSString *secureCardDescription;
/*! @brief This property represents the secure card retrieved. */
@property CoreSecureCard *coreSecureCard;
/*! @brief Converts object to NSDictionary. */
-(NSMutableDictionary*)getAsJsonObject:(CoreSecureCardResponse *) response;

@end
```

5 Glossary

URL	Universal Resource Locator
POS	Point of Sale
CVV	Card Verification Value
IOS	Formerly iPhone Operational System
OCR	Optical Character Recognition