Dotpay Technical Support
Wielicka Str. 28B, 30-552 Cracow, Poland
phone. +48 12 688 26 00
e-mail: tech@dotpay.pl

Worldwide secure payment

SDK (iOS)

Version 1.4.x

TABLE OF CONTENTS

INTRODUCTION

This document describes a set of software development tools (SDK library) that allows integrating merchant's mobile application with Dotpay payment system.

Thanks to devolving as many steps from our web application as possible to mobile application, the payment process is more convenient for a user and a developer obtains more control over it.

There is an option to modify visualization style of SDK (colors, fonts) to have best integration possible with merchant's mobile application.

The SDK library was created in Objective-C. It supports iOS system version 8.0 and higher.

This document uses the following terms and symbols:

Contractor / Merchant	Dotpay service user receiving the payment or owner of web shop, web page, on which payment process starts.
Shop	Merchant's web shop that uses mobile application.
Client / Buyer	The person making the payment to the merchant via the online transaction with a use of mobile application.
Developer	A developer, who creates mobile application for a merchant.

Related documents

Dotpay technical manual – a document that describes a basic payment process for web shops, available for downloading in Dotpay panel.

Getting started with SDK

You have to add the SDK to a project and initialize it in a proper way to use it. Details of these steps were described in next chapters.

In order to make it easier to start with SDK, we deliver a test application, available in `example` sub-directory.

Project settings

1. The Dotpay framework has to be added to a project. It is located in `lib` sub-directory (in the bookmark `Build Phases` please select section `Embed Frameworks`).
2. The JavaScriptCore framework has to be added to the project.
3. The header file has to be imported to every place that uses SDK:

```
#import <DotPaySDK/DotPaySDK.h>
```

Language settings

The SDK library, in a current version, supports Polish and English language. Further language extensions are planned so that dynamic configuration is recommended. In order to do this, you need to:

1. Download available languages list from z singleton DotPay:

```
 [[DotPay sharedInstance] getLanguageListWithCompletion:^(NSArray
*languageList, NSError *error){}];
```

2. Select most appropriate language
3. Set a selected language

```
[DotPay sharedInstance].defaultLanguage = bestLanguage;
```

System selection

The SDK library may communicate both with Dotpay production and test environment. SDK library works in production version by default.

In order to select the test system, you need to follow the instruction:

```
[DotPay sharedInstance].debugMode = YES;
```

In order to select the production system, you need to follow the instruction:

```
[DotPay sharedInstance].debugMode = NO;
```

SDK version

We recommend to display the SDK version in the web shop which will make problems diagnostic process easier in the future. SDK version can be downloaded from given Dotpay singleton property:

```
[DotPay sharedInstance].sdkShortVersionString
```

Deleting unnecessary architectures from Dynamic Libraries in Xcode

SDK has been created for all architectures so it can be used on devices and simulators. When building application with SDK for App Store unnecessary architectures have to be deleted from the project. To do so, after embed frameworks step add „run script":

```
APP_PATH="${TARGET_BUILD_DIR}/${WRAPPER_NAME}"

# This script loops through the frameworks embedded in the application and
# removes unused architectures.
find "$APP_PATH" -name '*.framework' -type d | while read -r FRAMEWORK
do
    FRAMEWORK_EXECUTABLE_NAME=$(defaults read "$FRAMEWORK/Info.plist"
CFBundleExecutable)
    FRAMEWORK_EXECUTABLE_PATH="$FRAMEWORK/$FRAMEWORK_EXECUTABLE_NAME"
    echo "Executable is $FRAMEWORK_EXECUTABLE_PATH"

    EXTRACTED_ARCHS=()

    for ARCH in $ARCHS
    do
        echo "Extracting $ARCH from $FRAMEWORK_EXECUTABLE_NAME"
        lipo -extract "$ARCH" "$FRAMEWORK_EXECUTABLE_PATH" -o
"$FRAMEWORK_EXECUTABLE_PATH-$ARCH"
        EXTRACTED_ARCHS+=("$FRAMEWORK_EXECUTABLE_PATH-$ARCH")
    done

    echo "Merging extracted architectures: ${ARCHS}"
    lipo -o "$FRAMEWORK_EXECUTABLE_PATH-merged" -create "${EXTRACTED_ARCHS[@]}"
    rm "${EXTRACTED_ARCHS[@]}"

    echo "Replacing original executable with thinned version"
    rm "$FRAMEWORK_EXECUTABLE_PATH"
    mv "$FRAMEWORK_EXECUTABLE_PATH-merged" "$FRAMEWORK_EXECUTABLE_PATH"

done
```

Payment process

The payment process consists of displaying controller `DPDotPayViewController` and awaiting calling out a delegate method that returns control.

The SDK library will guide the client through the process of selecting payment channel, passing on/verification client's data, selecting extra options, accepting proper terms of use and finalizing the payment process.

In the event of paying for real goods, payment status received from SDK is just informational, a right order status will be delivered in the backend system according to *Dotpay technical manual*.

In next chapters there are described steps required to use payment controllers and extra options that enable adjusting payment process to own needs.

Preparing return delegate

Preparing payment process starts from preparing delegate `DPDotPayViewControllerDelegate`:

```
-(void) dotpayViewController: (DPDotPayViewController *)viewController
didFinishPaymentWithSummary: (DPPaymentSummary *)paymentSummary;

-(void) dotpayViewController: (DPDotPayViewController *)viewController
didFailToFinishPaymentWithError: (NSError *)error
```

Initializing payment process

A payment description has to be prepared to start the payment process. An instance of a class `DPPaymentInfo` has to be created and filled out with data described in the table below:

PARAMETER	DESCRIPTION
`merchantID`	Type: `NSString` Merchant's ID number in Dotpay system
`amount`	Type: `NSDecimalNumber` Order amount
`textDescription`	Type: `NSString` Order description
`currency`	Type: `DPCurrency`

	Default value: „`PLN`" It describes currency of parameter amount. In the chapter Available currencies we,described a way how to download available currency list
`senderInformation`	Type: `NSDictionary` Default value: `nil` Additional information on client, e.g. name, surname, email…,Keys:,"firstname"; "lastname"; "email"; "phone"; "street"; "street_n1" – building number; "street_n2" – flat number; "postcode"; "city"; "country" (3 letters ISO3166). These values are not mandatory. We recommend to pass on at least name, surname and email. The payment form should be filled out with that type of data. SDK will ask a client for missing data. Specific explanation of these fields is described in *Dotpay technical manual*.
`additionalInformation`	Type: `NSDictionary` Default value: `nil` Extra parameters handed over in a payment process in accordance with additional technical manuals.
`control`	Type: `NSString` Default value: `nil` The parameter that defines a payment, handed over in payment confirmation, which is sent to a Shop. This parameter is required to match a payment status to an appropriate order in a Shop. More information you will find in the *Dotpay technical manual*. If not set, it's generated by SDK. ATTENTION To have properly working payment history, this parameter should be unique for every order.
`urlc`	Type: `NSString` Default value: `nil` URL address used for receiving payment information (order completed or rejected). More information you will find in the *Dotpay technical manual*.

Example:

```
DPPaymentInfo *info = [[DPPaymentInfo alloc] init];
info.currency = [DPCurrency currencyWithCode:@"PLN"];
info.amount = [NSDecimalNumber decimalNumberWithString:@"100.0"];
info.merchantID = @"10000";
info.textDescription = @"Some description";
info.senderInformation = @{@"firstname": @"John", @"surname": @"Doe", @"email":
@"john.doe@test.pl"};
info.additionalInformation = @{@"id1": @"12345", @"amount1": @"40", @"id1":
@"67890", @"amount2": @"60"};
```

The next step is downloading channels list by means of one of the following methods, passing on transaction details:

`[[DotPay sharedInstance] getChannelListForPaymentInfo:info withCompletion:^(NSArray *channelList, DPPaymentDetails *paymentDetails, NSError *error) { }]` – downloads all channels

`[[DotPay sharedInstance] getChannelListForPaymentInfo:info online:online withCompletion:^(NSArray *channelList, DPPaymentDetails *paymentDetails, NSError *error) { }]` – downloads channels that are currently online/offline

[[DotPay sharedInstance] getChannelListForPaymentInfo:info withIds:ids withCompletion:^(NSArray *channelList, DPPaymentDetails *paymentDetails, NSError *error) { }] – downloads channels according to ID

`[[DotPay sharedInstance] getChannelListForPaymentInfo:info group:group withCompletion:^(NSArray *channelList, DPPaymentDetails *paymentDetails, NSError *error) { }]` – downloads channels of given types

At the end a controller in the main thread is presented asynchronous and the following operations are executed:

a) Initializing payment controller with channels list and payment details

b) Optional last selected channel shutdown (this option is turned on by default)

c) Set a return delegate

d) Display a controller

Example:

```
dispatch_async(dispatch_get_main_queue(), ^{

    DPDotPayViewController *dotPayViewController = [[DPDotPayViewController
alloc] initWithPaymentChannelList:channelList paymentDetails:paymentDetails];
    dotPayViewController.useLastChannelSelection = NO;
    dotPayViewController.paymentControllerDelegate = self;

    [self presentViewController:dotPayViewController animated:YES
completion:NULL];

        });
}];
```

Finalizing payment process

A correctly finalized payment process is signalized by a proper delegate method. In the event of successful payment process, the method `dotpayViewController:didFinishPaymentWithStatus` is called out. The argument of this method is the class DPPaymentSummary object, that holds payment details.

If the payment process finishes with an error, the `dotpayViewController:didFailToFinishPaymentWithError`: method will be called out with argument, which is an error description.

Attention!!! A finalized payment process with a success doesn't mean the payment was processed, but it means the payment had no errors. A payment result will be returned in an appropriate event parameter.

Examples of event handlers:

```
-(void) dotpayViewController: (DPDotPayViewController *)viewController

didFinishPaymentWithSummary: (DPPaymentSummary *)paymentSummary {

        if([paymentSummary.status isEqualToString: kDPPaymentSummaryStatusCompleted])

{

             // payment successful

} else if ([paymentSummary.status isEqualToString: kDPPaymentSummaryStatusRejected]) {

             // payment rejected

        } else {

              //payment is being processed

       }

  }

-(void) dotpayViewController: (DPDotPayViewController *)viewController

didFailToFinishPaymentWithError: (NSError *)error {

             //payment error

  }
```

Summary details

On summary page it is possible to enable additional details like description, status and amount. Disabled by default.

In order to enable this functionality use method:

```
[[DotPay sharedInstance] setShowDetailsOnSummaryScreen:YES];
```

Available currencies

A list of currencies supported by Dotpay can be downloaded by the following method of PaymentManager:

```
 [[DotPay sharedInstance] getCurrencyListWithCompletion:^(NSArray *currencyList,
NSError *error){}];
```

Changing presentation style

In order to change presentation style of controls element in a payment process the `UIAppearance` has to be used.

Using the `appearanceWhenContainedIn` system control method, presentation changes may be limited only to specific payment controllers:

- `DPPaymentChannelsViewController` – channel selection controller
- `DPPaymentFormViewController` – Payment form controller. This controller is not available for developer, so that to use it the `NSClassFromString` is required `NSClassFromString(@”DPPaymentFormViewController”)`
- `DPPaymentSummaryViewController` – confirmation page controller

Example:

1. Change background color of `DPPaymentFormViewController` to red:

```
      [[UIView appearanceWhenContainedIn:
 NSClassFromString(@"DPPaymentFormViewController"), nil] setBackgroundColor:[UIColor
 redColor]];
```

2. Change color of active buttons in controler `DPDotPayViewController` to myCustomColor:

```
    [[UIBarButtonItem appearanceWhenContainedIn:[DPDotPayViewController class], nil]
setTintColor:myCustomColor];
```

3. Change literal text color from controller `DPDotPayViewController` to `myCustomColor`:

```
[[UILabel appearanceWhenContainedIn:[DPDotPayViewController class], nil]
setTextColor:myCustomColor];
```

It is also possible to change default status color using singleton table `DPStyleManager`:

```
[DPStyleManager sharedInstance].colorsForSummaryStatus = @{
        kDPPaymentSummaryStatusNew  : [UIColor greyColor],
        kDPPaymentSummaryStatusProcessing :[UIColor greyColor],
        kDPPaymentSummaryStatusRejected  : [UIColor redColor],
        kDPPaymentSummaryStatusCompleted  : [UIColor greenColor],
        kDPPaymentSummaryStatusProcessingRealizationWaiting : [UIColor greyColor],
        kDPPaymentSummaryStatusProcessingRealization : [UIColor greyColor]
    };
```

Changing summary screen styles

Using methods below you can change background color, title font, button font and color on summary screen:

- payment button color `(default is #ab191e)`

```
@property (nonatomic, strong) UIColor *paymentButtonColor;
- (void)setPaymentButtonColor:(UIColor * _Nonnull)paymentButtonColor;
```

- payment button text color `(default is white)`

```
@property (nonatomic, strong) UIColor *paymentButtonFontColor;
- (void)setPaymentButtonFontColor:(UIColor * _Nonnull)paymentButtonFontColor;
```

- screen header color `(default is #ab191e)`

```
@property (nonatomic, strong) UIColor *paymentHeaderDetailsColor;
- (void)setPaymentHeaderDetailsColor:(UIColor * _Nonnull)paymentHeaderDetailsColor;
```

- screen header text color `(default is white)`

```
@property (nonatomic, strong) UIColor *paymentHeaderDetailsFontColor;
- (void)setPaymentHeaderDetailsFontColor:(UIColor * _Nonnull)paymentHeaderDetailsFontColor;
```

Personal channel selection control

SDK library enables replacing default channel selection control, to adjust needs of Merchant's shop. The following steps are required to use this possibility:

1. Prepare an object of the `DPPaymentInfo` class and fill it out with data (in accordance with aforementioned chapter *Payment initialization*)

2. Download channels list (channels, in which we are not interested may be deleted) along with downloading payment details:

```
[[DotPay sharedInstance] getChannelListForPaymentInfo:info withCompletion:^(NSArray
*channelList,    DPPaymentDetails    *paymentDetails,    NSError    *error)    {    }
```

3. At the end a controller in the main thread is presented asynchronous and the following operations are executed:

   a. Initializing payment controller with channels list and payment details

   b. Initializing the `DPDotPayViewController` payment controller with indicating personal channel payment controller

   c. Set a return delegate

   d. Display a Dotpay payment controller

Example:

```
dispatch_async(dispatch_get_main_queue(), ^{
MyCustomChannelViewController *myChannelViewController =
[[MyCustomChannelViewController alloc] initWithPaymentChannelList:channelList ];
DPDotPayViewController *paymentViewController = [[DPDotPayViewController alloc]
initWithPaymentChannelSelectionController: myChannelViewController
paymentDetails:paymentDetails];
paymentViewController.paymentControllerDelegate = self;
[self presentViewController:paymentViewController animated:YES completion:NULL];
        });
    }];
```

Custom order summary controller

Library allows to change order summary controller with your own, better suited for shop's needs. In order to use it:

1. In delegate `DPDotPayViewControllerDelegate` disable order summary page.
2. Initializing the `DPDotPayViewControllerDelegate` payment controller with indicating personal channel payment controller and method

   ```
   - (BOOL)dotpayViewController:(DPDotPayViewController *)dotpayViewController
   shouldShowViewControllerForPaymentSummary:(DPPaymentSummary *)paymentSummary;
   ```

3. Setting a return delegate
4. Displaying Dotpay payment controller

Example:

```
- (BOOL)dotpayViewController:(DPDotPayViewController *)dotpayViewController
shouldShowViewControllerForPaymentSummary:(DPPaymentSummary *)paymentSummary
    {
         return NO;
    }
(void)dotpayViewController:(DPDotPayViewController *)dotpayViewController
didFinishPaymentWithSummary:(DPPaymentSummary *)paymentSummary {
    MySummaryViewController *controller = [[MySummaryViewController alloc]
initWithPaymentSummary:paymentSummary];
    dispatch_async(dispatch_get_main_queue(), ^{
[self presentViewController:controller animated:YES completion ^{}];
    });
}
```

Special channels support

Extra functions related to special payment channels are described.

Credit card payment - 1Click

1Click functionality enables quick payment process with a saved credit card. Basic credit card data are saved in Dotpay system.

This functionality (if available in Merchant's shop) is turned on by default in SDK. A client's consent is also required to use 1Click (while filling out the payment form).

In order to turn off 1Click, the following command is required:.

```
[DPOneClickManager sharedInstance].enabled = NO;
```

ATTENTION

After turning off aforementioned options, formerly saved credit card data is not removed. To remove that data, the following commands are required. An additional Developer's duty is to create an option to remove saved credit card data.

Methods for developer

To register new card initialize object `DPOneClickPaymentCardInfo` and call method:

```
[[DPOneClickManager sharedInstance] registerPaymentCardWithCardInfo:cardInfo
withCompletion:^(DPOneClickPaymentCard *response, NSError *error)
```

In order to check which card is default call method:

```
[[DPOneClickManager sharedInstance] defaultOneClickPaymentCard];
```

To mark card as default call method:

```
[[DPOneClickManager sharedInstance]setDefaultOneClickPaymentCardWithId:(NSString
*)cardId];
```

To check what cards are remembered call method:

```
[[DPOneClickManager sharedInstance] paymentCards];
```

To delete card call:

```
[DPOneClickManager sharedInstance] unregisterPaymentCardWithId:<#(NSString *)#>
withCompletion:<#^(NSDictionary *response, NSError *error)completion#>];
```

To delete all remembered cards call:

```
[[DPOneClickManager sharedInstance] clearData];
```

Calling this method will delete all remembered cards in Dotpay and all data stored locally.

Managing cards

Additional feature of library is card manager. In manager all remembered cards are listed. Manager also allows to add cards, mark them as default and delete them.

Using built-in controller

In order to use built-in controller managing cards initiate and display controller `DPOneClickPaymentCardManagerViewController:`

```
NSString *merchantId = #yourMerchantId
NSString *currency = #yourcurrencycode //(eg. @"PLN")
DPOneClickPaymentCardManagerViewController *viewController =
[[DPOneClickPaymentCardManagerViewController alloc] initWithMerchantId:merchantId
currency:currency];
[self presentViewController:viewController animated:YES completion:nil];
```

Custom card management controller

To have data presented in a way better for shop, you can create own controller for card management using methods provided by library.

Paying with card without filling the form

Library allows to make payment with one click without displaying payment summary form.

ATTENTION

When making a payment it is possible to make additional card verification, independent from SDK.

To make a payment initialize object `DPPaymentInfo` and call method:

```
[[DotPay sharedInstance] oneClickPaymentWithPaymentInfo:paymentInfo paymentCard:nil
withCompletion:^(DPPaymentSummary *_Nullable jsonResponse, NSError * _Nonnull error)
```

ATTENTION

If parameter `paymentCard:nil` then payment will be made with default card.

External cards data

Cards references are remembered inside the SDK by default, so the functionality is available without any additional work necessary either on the side of the mobile application or backend with which the application can work. A side effect of this approach is the inability to transfer registered cards between devices or after reinstalling the application.

However, if the application works with its own backend, uniquely identifying the User (this is a necessary condition), card references can be stored in the backend, which will allow them to be shared (e.g. if the user account is shared between the website and the mobile application).

Detailed description about card storage mechanisms you can find at Technical manual for payments implementation Dotpay.

To enable support for such stored cards, `DPOneClickManager` should be initialized correctly:

```
[[DPOneClickManager sharedInstance]configureWithExternalCards:( NSArray <
DPOneClickExternalPaymentCard *> *)externalCards forCustomerId:( NSString
*)customerId];
```

Where `customerId` is equivalent to the `credit_card_customer_id` parameter described in the main technical integration documentation, the `externalCards` is a list of `DPOneClickExternalPaymentCard` objects containing complete information about the card. Description of properties in the table below:

PARAMETER	DESCRIPTION
maskedNumber	Type: NSString Masced card number
creditCardId	Type: NSString Credit card ID (according to the Dotpay basic documentation)

brandName	Type: NSString Card name
brandCodename	Type: NSString Card Type (according to the Dotpay basic documentation)
logoUrlPath	Type: NSString Link to the card's logo (according to the Dotpay basic documentation, the link is returned together with the data of registered card)

ATTENTION

the method of setting cards from the outside overwrites all cards added previously marked as external, this method can be called repeatedly to update the list of cards.

If we want to remove the user context, remove the added cards from the SDK memory so as to work again only in the context of the cards remembered in the SDK, call method:

```
[[DPOneClickManager sharedInstance] clearExternalCardsData]
```

In addition, it is possible to disable the visibility of cards remembered inside the SDK (if the application currently uses such cards, or if it must work both in a context without a logged in user / with a logged in user). To disable the use of such cards, just set the following property:

```
 [DPOneClickManager sharedInstance].useInternalCards = NO;
```

1. Card registration in the context of an external User

When initializing the SDK with external card data, note that all cards registered from the SDK level will be registered with the given `credit_card_customer_id` (both by the Card Manager and those registered during payment).

If the User has no registered card yet, but we want to register new cards in its context, the If the User has no registered card yet, but we want to register new cards in its context, the `configureWithExternalCards` method should be called with an empty list of cards.

All cards registered in this way are automatically added to the SDK, as like external cards (in the context of the `indicated credit_card_customer_id`). Information about the registration of a new card can be obtained at the backend level. As soon as the application receives this information, you must reinitialize `DPOneClickManager` using current list of external cards.

Receive information about changes caused by User actions

To obtain information about the impact of User actions on the database of external cards (about the fact of registration, changing the default card), you can implement the DPOneClickManagerDelegate delegate:

```
- (void) oneClickManager:( DPOneClickManager *)manager didRegisterNewCreditCard:(
DPOneClickCardRegistrationData *)cardRegistrationData;
- (void) oneClickManager:( DPOneClickManager *)manager
didChangeDefaultOneClickPaymentCard:( DPOneClickPaymentCard *)newDefaultCard;
```

And set a reference :

```
[DPOneClickManager sharedInstance].delegate = self;
```

As soon as a new card is registered, the application will be notified about that by calling the didRegisterNewCreditCard method of the implemented delegate, so the application can send a request to backend for refresh the list of cards.

When the default card changes (please note that the default card is a different concept from the one last used for standard payment), the `didChangeDefaultOneClickPaymentCard` method of the implemented delegate will be called.

Masking the cvv filed

If the SDK is used in an application that is often used by Users in conditions that limit the possibility of maintaining an adequate level of confidentiality of entered data (e.g. purchase of a public transport ticket after entering a crowded vehicle), you can use the field masking option on the CVV to give The user has a greater sense of security.

To do this call the method:

```
[DPOneClickManager sharedInstance].treatsCvvAsPassword = YES;
```

However, it should be noted that this value is only one of the factors ensuring the security of card payments, hence there is no recommendation that this field should be so secured in every application, remembering that incorrect entry of this field leads to unsuccessful payments, and multiple unsuccessful payments can lead to blocking the card for online payments.

Payment without the payment form.

In case you need to remove a special payment channel to the basket of the mobile application (e.g. Visa Checkout).It was created mechanism that allows calling the payment with the channel indication, without the option to choose the payment channel and payment form.

To make such a payment, initialize the `DPPaymentInfo` object and call the method below:

```
 [[DotPay sharedInstance] fastPaymentWithPaymentInfo:paymentInfo paymentChannelId:
channelId showingDefaultSummary:defaultSummary withCompletion:^(DPPaymentSummary
*_Nullable jsonResponse, NSError * _Nullable error))completion;
```

The parameter specifying the channel identifier is mandatory. Additional parameters specific to the channel should be send as elements of `additionalInformation` dictionary in the `DPPaymentInfo` object.

Masterpass Champion Wallet

Masterpass Champion Wallet allows you to quickly and conveniently use the Masterpass payment functionality when you do not have such a wallet during payment. Registering Masterpass Champion Wallet is convenient, it can be carried out completely online and allows for "1click" payments. After registering / logging into the wallet once, subsequent payments are carried out with the minimum amount of information.

To use the functionality, after business arrangements, an appropriate configuration of the Store account is performed, which automatically gives access to this version of the wallet

ATTENTION

to use this functionality, mobile applications using the SDK must have a dedicated Dotpay Store account.

Additional parameters for payment initiation

To improve the process of using the wallet by the Payer, it is recommended to pass the `phone` parameter in the `senderInformation` property of the `PaymentInformation` object, with which the SDK instance is initiated.

In addition, if the `phone_verified` parameter with the value "true" is passed, the transfer of which the mobile application clearly confirms that the phone number transferred to the SDK is fully verified (e.g. with a single-use SMS code), the registration process will be simplified.

Payment authorization

Masterpass Champion Wallet was created to make payments as easy as possible. To reduce the risk of frauds, it is recommended to perform payment authorization on the application side. To do this, you need implement the `DPMasterpassPaymentDelegate` delegate:

```
-(void) masterpassPaymentShouldCreateAuthorizationOn:(UIViewController
<DPMasterpassAuthorization> *)viewController;
- (void)masterpassPaymentShouldAuthorizeUserPaymentOn:(UIViewController
<DPMasterpassAuthorization> *)viewController;
```

And set it on the main SDK controller, e.g .:

```
dotPayViewController.masterpassPaymentDelegate = self;
```

Passed in parameters specified methods UIViewKontroler implements the delegate methods `DPMasterpassAuthorization`:

```
- (void)didCreateAuthorization:(BOOL)successful andError:(NSError *)error;
- (void)didAuthorizeUser:(BOOL)successful andError:(NSError *)error;
```

to be used for the transfer of control.

`masterpassPaymentShouldCreateAuthorizationOn` methods will be called during the user logs in (or registers) to the wallet, and allows to define / initiate authorization mechanisms in the mobile application (e.g. to define the application PIN). After completing this process, you should call the `authorizationCreateResult` method by passing the authorization creation status. In the event of a positive one, the SDK will continue the registration process to the wallet, in the case of a negative one it will terminate this process.

The `onPaymentShouldAuthorizeUser` method will be called whenever the logged-in Payer wants to make another payment with the associated wallet. As part of its course, we should perform authorization (e.g. PIN verification), and when finished, call the `didAuthorizeUser` method of delegate, transferring its status. Confirmation of authorization will allow you to make a payment, information about incorrect authorization will interrupt it.

Example:

```
-(void)                masterpassPaymentShouldCreateAuthorizationOn:(UIViewController
<DPMasterpassAuthorization> *)viewController {

        [viewController didCreateAuthorization:YES andError:nil];

    }

-(void)        masterpassPaymentShouldAuthorizeUserPaymentOn:        (UIViewController
<DPMasterpassAuthorization> *)viewController {

        [viewController didAuthorizeUser:YES andError:nil];

}
```

Transaction history and status

SDK library offers saving and displaying transaction history. The transaction history also displays related operations, e.g. subsequently made refunds and other additional operations. Additional operations from history data are also available.

Using built-in controller

The use of built-in control In order to use built-in history control, the `DPPaymentHistoryController` controller has to be initialized and displayed:

```
DPPaymentHistoryController *historyController = [[DPPaymentHistoryController alloc]
initWithPaymentHistory:nil];
[self presentViewController:historyController animated:YES completion:NULL];
```

Changing presentation style

A change of presentation way of controls elements in payment process is made with a use of the `UIAppearance` framework. Detailes are described in the chapter *Changing presentation style.*

A history controller, for which presentation change limitations may be applied to is `DPPaymentHistoryController`.

Example:

```
[[UIView appearanceWhenContainedIn: [DPPaymentHistoryController class], nil]
setBackgroundColor:[UIColor redColor]];
```

Personal history control

Personal history control SDK library offers creating personal history control by downloading only data from the library, to better match web shop specification.

Downloading history data (table of objects of class `DPPaymentSummary`) is available in property `paymentHistory` singletona `DPPaymentHistory`:

```
NSArray *paymentSummaries = [DPPaymentHistory sharedInstance].paymentHistory;
```

Additionally for transactions in personal history you can:

1. Remove a single transaction:

```
[[DPPaymentHistory sharedInstance] removePaymentSummary: paymentToRemove];
```

2. Check current payment status for a record in the history and refresh displayed data:

   a. In personal view `DPPaymentHistoryObserver`, observer methods are implemented, which refresh history view:

   ```
   - (void)paymentHistory:(DPPaymentHistory *)paymentHistory
   didUpdatePaymentSummary:(DPPaymentSummary *)paymentSummary;
   - (void)paymentHistory:(DPPaymentHistory *)paymentHistory
   didFailedToUpdatePaymentSummary:(DPPaymentSummary *)paymentSummary
   withError:(NSError *)error;
   ```

   b. Register observer waiting for history changes:

   ```
   [[DPPaymentHistory sharedInstance] registerObserver: self]
   ```

   c. Call out a method that updates payment status:

   ```
   [[DPPaymentHistory sharedInstance] updatePaymentSummary: paymentToUpdate];
   ```

   d. While releasing custom history controller, unregister history change observer:

   ```
   [[DPPaymentHistory sharedInstance] unRegisterObserver: self]
   ```