

**Manual**

**DoubleClue Developer’s Guide (iOS)**

**Version: 2.1.0**

Contents

[1 Introduction 3](#_Toc501439683)

[1.1 Disclaimer 3](#_Toc501439684)

[1.2 App Structure 3](#_Toc501439685)

[1.2.1 App GUI 4](#_Toc501439686)

[1.2.2 SDK-Library 4](#_Toc501439687)

[1.3 Requirements 4](#_Toc501439688)

[1.4 Contents of Deliverables 4](#_Toc501439689)

[2 DoubleClue Enterprise Management (DCEM) Files 5](#_Toc501439690)

[2.1 “TrustStore.pem” 5](#_Toc501439691)

[2.2 “SdkConfig.as” 5](#_Toc501439692)

[3 Getting Started with “LibDoubleClue” 7](#_Toc501439693)

[4 Using “LibDoubleClue” 8](#_Toc501439694)

[4.1 Initialising “DCAppSdk” 8](#_Toc501439695)

[4.2 Making Calls to the Server 9](#_Toc501439696)

[4.3 Messages 11](#_Toc501439697)

[4.3.1 Receiving Messages 11](#_Toc501439698)

[4.3.2 Responding to Messages 11](#_Toc501439699)

[4.3.3 Timeout Messages 12](#_Toc501439700)

[4.4 Example 12](#_Toc501439701)

[4.5 Cloud Data 15](#_Toc501439702)

[4.5.1 Push Notifications 15](#_Toc501439703)

[4.6 Error Handling 15](#_Toc501439704)

# Introduction

## Disclaimer

This guide is intended for developers who already possess some knowledge of iOS development with Xcode and Swift / Objective-C. The sample GUI code provided in the Release Package is written in Swift 4.

With the DoubleClue iOS App, users are able to register at and login to a DoubleClue Enterprise Management (DCEM) server. After logging in, users can receive messages and send back signed message responses.

## App Structure

The DoubleClue iOS App consists of two main parts:

* App GUI
* DoubleClue Software Development Kit (SDK) Framework (LibDoubleClue)

**App GUI**

Open Source

**DoubleClue SDK Framework**

(Security & Communication)

**DC SDK-API**

In our sample code, we also have an additional layer built into the GUI source code – a wrapper class for the underlying framework’s SDK API called **DCWrapper**. This is written in Swift, and we would recommend using it if you are using Swift for your application. It exposes all necessary methods and callbacks from the framework, and converts data into more Swift-friendly objects.

### App GUI

The App GUI is the iOS App itself, including all user interfaces and GUI navigations. This is a paradigm of a typical DoubleClue App, an example of which is delivered in source files. You are free to change the sources according to your needs after signing the DoubleClue License agreement.

### SDK-Library

The SDK-Library is responsible for all cryptography and secure communication with DCEM. The library is delivered as **LibDoubleClue.framework**. The interface between the App GUI and the SDK-Library is the DoubleClue SDK-API, written in Objective-C. A Swift wrapper is also available in our sample code (**DCWrapper.swift).**

## Requirements

* Xcode v9.0 or later.
* An Apple device or Xcode Simulator which can run iOS 10.0 or later. The framework can be used for earlier versions of iOS, but the sample GUI app is built for iOS 10.

## Contents of Deliverables

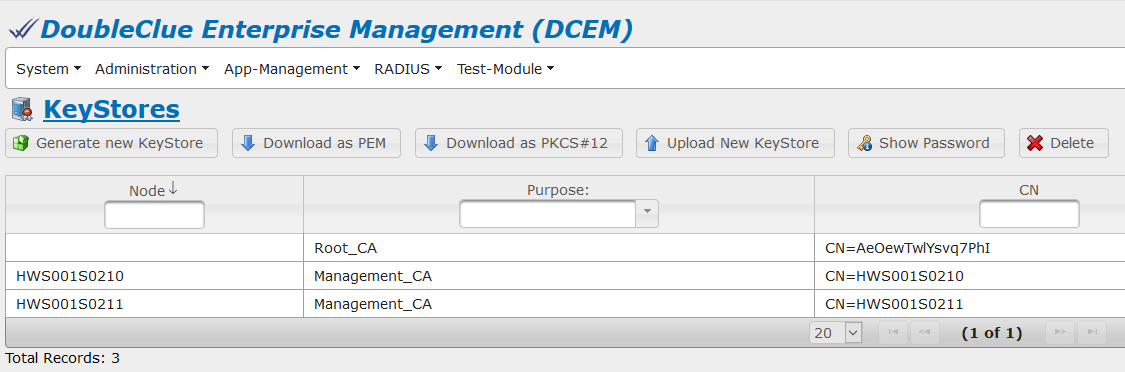
* “**LibDoubleClue**” – A folder containing two versions of “**LibDoubleClue.framework**”. They are static libraries containing the client-side functionality for DoubleClue. The smaller Release Version is intended for release versions of the app since it only supports physical Apple devices. The larger Development Version supports the Xcode Simulator and is meant for development versions of the app.
* “**DC\_iOS\_Dev\_Guide.docx**” – This guide is meant to be used by developers who want to create an app using DoubleClue authentication or include it in their own iOS project.
* “**Source**”– A folder containing the source files for our sample DoubleClue GUI app. This code will be referenced in this document. Please open the “**xcworkspace**” file, not the “xcodeproj” file.

# DoubleClue Enterprise Management (DCEM) Files

In order to create a secure connection to DCEM, you will need two files from the DCEM installation, “**SdkConfig.as**” and “**TrustStore.pem**”. These files have to be copied somewhere inside the GUI app’s project directory, then included in the Xcode project itself.

## “TrustStore.pem”

This file contains the DCEM Root-Trust X.509 certificates in PEM format. These are required to establish a secure SSL/TLS connection to DCEM or the DCEM Load Balancer. If SSL/TLS is terminated at the Load Balancer, you have to retrieve the file from the Load Balancer, else you need to download this file from DCEM (main menu “System”, sub menu “KeyStores”, choose a ‘DeviceWebsockets\_CA’, button “Download as PEM”).



For more information please have a look at the DCEM Manual (“**DCEM\_Manual\_EN.pdf**”) or ask your DCEM administrator.

## “SdkConfig.as”

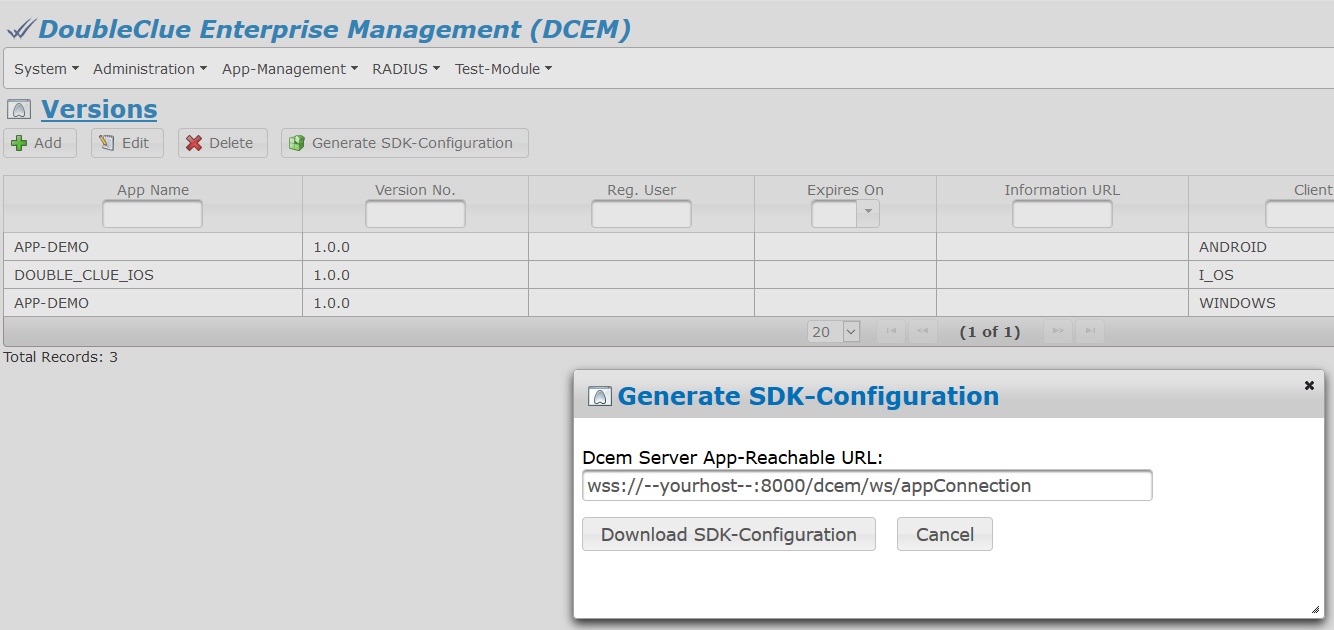
C:\Users\maike.behnsen\Desktop\Grafiken\caution-sign.pngThis file contains several secure information which are required to establish a trust connection to your DCEM installation: the Server URL, the public-Key of the trust DCEM and other secure information.

The file is signed and any changes made in it will corrupt the file contents.

Note: DoubleClue uses an extra X.509 trust certificate infrastructure between SDK-Clients and DCEM, which is independent from the SSL/TLS certificates.

The file has to be downloaded from DCEM (main menu “App-Management”, sub menu “Versions”, button “Generate SDK-Configuration”).

For more information please have a look at the DCEM Manual (“**DCEM\_Manual\_EN.pdf**”) or ask your DCEM administrator.

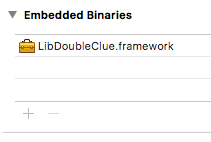


# Getting Started with “LibDoubleClue”

1. Copy one of the versions of “**LibDoubleClue.framework”** into your project directory. The version you choose depends on the platform you want to build for. The Development version supports the Xcode Simulator, while the Release version does not.
2. Include the file into your Xcode project.



1. Go to your project settings, select your “**Target**” and go to the “**General**”tab**.**
2. The framework might have already been added to “**Linked Frameworks and Libraries**”. In this case, remove it by clicking the “**-**” button underneath (otherwise it will be added there twice).
3. Drag the framework file you have just included into the “**Embedded Binaries**”.



The framework is now ready for use. If you are using **Swift**, simply add

import LibDoubleClue

to the beginning of a class which will use DoubleClue, and it will provide you with all exposed functionality of the framework. If you are using **Objective-C**, you will need to import the necessary header files for the functionality you would like to use, such as

#import <LibDoubleClue/DCAppSdk.h>

To get the sample code to work, you only need to follow steps 1-5. The app should then be able to build and run successfully. Note that the Release Version will not work if you try to build it for the Simulator.

# Using “LibDoubleClue”

## Initialising “DCAppSdk”

“**DCAppSdk**” is the main class used for communicating with a DoubleClue server. A *single* instance of it should exist throughout the entire lifetime of an app.

Locally, it stores a list of users registered with the current device, and can also perform offline functions, such as generating passcodes. Its main function however, is to send encrypted requests to a server and receive callbacks from it asynchronously.

To use “DCAppSdk”, it must first be initialised via the method:

-(void) initializeWithVersion: (DCVersion \*) version

delegate: (id) dcAppSdkDelegate

sdkConfig: (NSData \*) sdkConfig

properties: (NSMutableDictionary \*) properties

trustStore: (NSData \*) trustStore

error: (NSError \*\*) error;

Please refer to the in-line documentation for technical details of the parameters. An example is also provided in the sample code, in DCWrapper.initialize(). Note that the above is the Objective-C structure of the function, but the sample code uses a Swift version of it.

The parameters “**version**” and “**properties**” should contain information that match an entry in DCEM’s “**Versions**” page.

The parameters sdkConfig and trustStoreshould contain data retrieved from “**SdkConfig.as**” and “**TrustStore.pem**”. These files should be included in the solution. They are downloaded from DCEM to the server that the app should connect to (see Section 2).

The dcAppSdkDelegate should be an object which subscribes to the protocol called “**DCAppSdkDelegate**”. This will be the object which “DCAppSdk” will call should it receive callbacks from the server.

## Making Calls to the Server

There are a variety of calls that “DCAppSdk” can send to the server. However, they all follow the same pattern.

First, the call is made to the server via a function of DCAppSdk, such as

var error: NSError?

appSDK.login(withUserID: "HWS", password: "1234", error: &error)

whereappSDK is a previously initialized instance of “DCAppSdk”.

The function might fail, in which case no call would actually be sent. This can be identified by the parameter error, which should be checked. If it remains “**nil**”, then the call is successful.

The function itself does not return anything. Instead, the object awaiting the response should implement the function’s counterpart callback function, as per the protocol “**DCAppSdkDelegate**” (please refer to the table on the next page). Following on the previous example, this would be something like the following simple example:

func onLogin(\_ response: DCLoginResponse!, \_ error: NSErrorPointer) {

guard (error != nil) else {

NSLog("Login failed!")

return

}

NSLog("Logged in successfully!")

}

Implementations should check the error parameter for any failures that might have happened in the request (see Section 4.6). Each function may send other data with the callback. In this case, it sent an instance of DCLoginResponse, which may contain some data such as the expiration date of the current version of the app.

In our sample code, we created a more Swift-friendly wrapper for these functions, which we called “**DCWrapper**”. This structure is not required, however we would recommend using this wrapper for Swift projects, as it also converts the errors into more readable versions for Swift.

The following table lists DCAppSdk’s calls together with their counterpart callback methods from “**DCAppSdkDelegate**”.

|  |  |
| --- | --- |
| **DCAppSdk Function** | **Callback Function** |
| login | onLogin |
| disconnect | onDisconnect |
| activate | onActivation |
| sendMessageResponse | onSendMessageResponse\* |
| changePassword | onChangePassword |
| deactivate | onDeactivation |
| requestActivationCode | onActivationCodeResponse |
| verifyPassword | onVerifyPasswordResponse |
| sendQrCodeLogin | onSendQrCodeLoginResponse |
| getMapData | onGetMapDataResponse |
| setMapData | onSetMapDataResponse |

\*onSendMessageResponse is the immediate callback to sendMessageResponse, however, objects implementing the callback should also implement the following two functions:

* onReceive – This will send the original HTML template message to send a response to, and any subsequent messages from the server.
* onReceiveMessageTimedOut – This will be sent if the user takes too long to respond to a message, and its assigned time limit runs out.

“DCWrapper” responds to all available callback functions, and sends parsed responses to its own delegate. Please refer to the code to see which callback calls which delegate function.

“DCAppSdk” also has some offline functions, such as initialize, getInfo, getActivatedUsers and getAppPasscodeWithError. These functions are synchronous and do not have a callback, so their results are retrieved immediately from their respective return values (if any).

## Messages

One of the central features of DoubleClue authentication is the sending of messages from a server to a user via platform-independent HTML templates (as configured in DCEM). An app built with the intention of using “LibDoubleClue” should have the ability to render such templates and send responses back to the server.

### Receiving Messages

HTML messages are received via the “**DCAppSdkDelegate**” function onReceive. One of the parameters is an instance of DCReceivedMessage, whose property content is a string containing HTML code which defines the template to be shown. In our sample code, we are using an instance of WKWebView to render the messages, as seen in our custom UIView called “**HTMLView**”.

### Responding to Messages

In order to respond to these messages, it is necessary to use DCAppSdk.sendMessageResponse and pass an instance of DCResponseMessage to it. This should contain the same msgId as the previously received DCReceivedMessage. It is also necessary to include one of the error codes in “**DCErrorCodes**” within the property errorCode:

* OK if there were no errors encountered locally.
* The adequate error code if a problem has occurred, such as a timeout (APP\_MSG\_TIMEDOUT) or invalid input (INVALID\_INPUT\_PARSING).

Finally, if the error code is OK, it is necessary to populate the following two properties:

* actionId – A string identifying the button that was interacted with.
* responseData – A dictionary containing information about the user to send to the server. This may be empty if the template does not require the user to enter data. If it does, the dictionary keys should contain IDs of the HTML input elements, while the corresponding values should contain the entered data.

C:\Users\maike.behnsen\Desktop\Grafiken\caution-sign.pngNot all messages require a response. This can be checked with the received message’s property responseRequired, a boolean indicating whether one is needed. Responding to a message that does not require a response will result in an error.

An example of message responding can be found in our sample code, in DCWrapper.sendMessageResponse. Our method of retrieving the data we need for the response is via a Javascript source (see HTMLView.jsSourcePrepareButtons) which lists all inputs into a JSON string. This is then parsed into a dictionary and sent in the response.

### Timeout Messages

Messages that require a response are assigned with a time limit within which they expect that response. Therefore, there is a possibility that the server sends a message to the user that the message timed out, before the user triggers the expected response.

When this happens, “DCAppSdk” triggers the “**DCAppSdkDelegate**” function onReceiveMessageTimedOut, so this function needs to be implemented by any class which is expecting HTML messages from the server. The app should then block the user from doing any subsequent inputs to the message, and send a message response automatically with the error code APP\_MSG\_TIMEDOUT.

It is important to send this message response, as it allows the server to close the message cleanly with a logged error code and store it in the database.

## Example

Here is a simple example to illustrate the receiving of HTML messages and the sending of responses to them. Please note that this example is written in Swift 4.

First, a message is received from the server. This triggers the “**DCAppSdkDelegate**” function onReceive. A class in the app must implement this function and read the HTML data from it

var pendingMessage: DCReceivedMessage?

func onReceive(\_ receivedMessageValue: DCReceivedMessage!) {

pendingMessage = receivedMessageValue

renderHTML(receivedMessageValue.content)

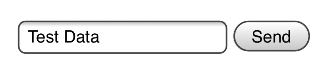
}

where renderHTML is some function which can load the HTML on-screen. Note that the message is saved locally – this is because it will be needed later on. Let us say that the HTML content in this case is the following:

<input type="text" id="inputData" /><button id="sendAction">Send</button>

Note that each element has an ID value. These values will be used in the response; the button’s ID will be the actionId and the text-field’s ID will be the key for its value.

Let us say the user types in some text (“Test Data”) into the text field and clicks the “Send” button.



The app should capture this event and call the server to send the data:

func sendResponse(errorCode: String?) throws {

let response = DCResponseMessage()

if let receivedMessage = pendingMessage, receivedMessage.responseRequired {

response.msgId = receivedMessage.msgId

if let previousError = errorCode {

response.errorCode = previousError

} else {

let responseData = NSMutableDictionary()

// textFieldValue is "Test Data", textFieldID is "inputData"

responseData.setValue(textFieldValue, forKey: textFieldID)

response.errorCode = OK

response.actionId = buttonID // "sendAction"

response.responseData = responseData

}

} else {

response.errorCode = INVALID\_STATE

}

pendingMessage = nil

let errorPointer: NSErrorPointer = nil

appSDK.sendMessageResponse(response, error: errorPointer)

if let error = errorPointer?.pointee {

throw error

}

}

appSDK is a previously initialized instance of “**DCAppSDK**”. This example is simplified such that the response data initialization is not dynamic. In an actual implementation, it is recommended that there is a system which checks for any input data within the HTML and populate the dictionary dynamically, so that it can handle a variety of templates from the server. Our sample GUI code contains such functionality (see DCWrapper.sendMessageResponse).

Since we have called the function sendMessageResponse, we need to implement the callback for it:

func onSendMessageResponse(\_ error: NSErrorPointer) {

if let errorInfo = error?.pointee?.userInfo {

NSLog("Message sending failed: %@", errorInfo[EXCEPTION\_SECOND\_CODE] as! String)

} else {

NSLog("Message sent successfully!")

}

}

This completes the cycle of receiving and sending messages. However, we need to handle the case that the server might send a timeout message before the user has triggered an action.

You might have noticed that the function sendResponse described above has an optional String parameter called errorCode. If it contains a value, this causes the response message to send the specified error code. This is for cases where a previous error occurred locally, such as an invalid input.

Therefore, we can use it in an implementation of the timeout callback function:

func onReceiveMessageTimedOut() {

do {

try sendResponse(errorCode: APP\_MSG\_TIMEDOUT)

} catch {

NSLog("Timeout response failed")

}

}

## CloudSafe

Another feature of DCEM is its ability to store and retrieve custom data on the server, which is adequately named “**CloudSafe**”. The type of data to store is up to the discretion of the developer. It can be anything from Strings to entire binary files. Data is stored as an encrypted byte array, and is associated with a String key.

DCAppSdk’s methods “setCloudSafe” and “getCloudData” handle the functionality of setting and getting Cloud data respectively. Data comes as an instance of DCCloudData, whose property content contains the actual data, while key contains the String key. Please take care to implement the adequate callbacks if you are using these functions (see Section 4.2).

### Push Notifications

DCEM has in-built functionality which allows it to use a special instance of Cloud Data in order to send Push Notifications to devices using DoubleClue authentication, such as informing them that they received a message while their app was closed.

DCEM uses Google’s *Firebase* as a Cloud messaging service. In order to connect with devices, these need to upload a Firebase token to DCEM via an instance of DCCloudData. The key in this case needs to be set as “hws.pushnotification.token”.

You can find a guide on how to set up Firebase in your app via Google’s own [documentation](https://firebase.google.com/docs/cloud-messaging/ios/client). You can also refer to our sample GUI code for this setup (see **AppDelegate**) and how to send the token to the server (see **FirebaseTokenManager**).

## Error Handling

Errors reported by LibDoubleClue come in the form of error pointers (NSErrorPointer in Swift, or NSError\*\* in Objective-C).

As is traditional with Objective-C, DCAppSdk’s functions require that an error is passed by reference, whose contents then need to be checked after the call (although in Swift, some of these functions are converted to throw errors instead). For example:

var error: NSError?

appSDK.verifyPassword("password", error: &error)

if let e = error {

NSLog("error with verifying password: %@", e.userInfo)

}

whereappSDK is a previously initialized instance of DCAppSdk.

Furthermore, all callbacks pass error pointers as parameters. These errors must be checked, as they determine whether a request was successful (error is “**nil**”) or not, in which case the cause of the error can be determined via error codes.

LibDoubleClue errors make use of the NSError property called userInfo. If an error occurred, this dictionary will be populated with some of the following keys (found in “**DCConstants**”).

* EXCEPTION\_CODE – Will always be populated.
* EXCEPTION\_SECOND\_CODE – Will only be populated if the error occurred in the server.
* EXCEPTION\_DETAILS – May be populated with further information about the error.

If EXCEPTION\_CODE exists while EXCEPTION\_SECOND\_CODE does not, then the error originated from LibDoubleClue. Otherwise, it originated from the server.

A full list of error codes can be found in the sample GUI code’s “Localization” files (in the workspace, these can be found under “Resources” > “Languages”). They can be identified via prefixes in the keys.

* **"error.ERROR\_CODE"** – These are errors originating from the DC SDK framework, where the error code would match the value of EXCEPTION\_CODE.
* **"error.server.ERROR\_CODE"** – These are server errors, where the error code would match the value of EXCEPTION\_SECOND\_CODE.
* **"error.gui.ERROR\_CODE"** – These are custom error codes for the GUI app. They do not correlate to any server / framework errors.

It is recommended that these localizations are copied into your project. Examples of their use may be found in our sample code; in DCWrapper.getDCWrapperError and DCRespondingViewController.getErrorMessage.