nets

Nets Denmark A/S
Lautrupbjerg 10
P.O. 500
DK-2750 Ballerup

T +45 44 68 44 68
F +45 44 86 09 30
www.nets.eu

CVR-nr. 20016175

NemID JavaScript Client Integration for Mobile Applications

nets

Document History

Version	Date	Edited by	Change/Reasons for Change/Comments
1.0	24-04-2016	PCKOC	First public version of document.
1.1	17-05-2016	PCKOC	Minor changes. Removing use of clientmode parameter, now only URL is used to decide clientmode, clientmode parameter is ignored. Text on SP test backend bug removed.
1.2	13-01-2017	DBKRA	Fix typos and link errors in SP-documentation for MobilApp
1.3	13-10-2017	JONCH	Adding a section on integrating App Switch. Also updated the section on Advanced Parameters.
1.4	26-10-2017	STSOR	Adding workaround for problem with entering into username field, using UIWebView on iOS.
1.5	06-03-2019	PCKOC	Introduced additions of Security best practice recommendations and explanations of the use of this in the app examples. Removed description for Windows Phone, as this is no longer supported.

Content

Document History ... 2
Content ... 3
1 Introduction ... 5
1.1 Document Scope ... 5
2 NemID JS Client integration into Mobile Applications ... 5
2.1 Parameter Generation ... 7
2.2 Response Handling ... 7
2.3 Iframe Integration and Running the Client ... 8
2.3.1 Limited Mode and Client Size ... 8
2.4 Integration of Remember User ID ... 10
2.5 Integration of App Switch ... 11
3 Android and iOS Source Example Packages ... 11
3.1 General Functionality of the Example Apps ... 11
3.1.1 Supported Flows ... 11
3.1.2 Functionality Overview ... 12
3.1.3 MainView Options and App/Backend functionality ... 14
3.1.3.4 Advanced Parameters ... 16
3.1.4 General App Limitations ... 16
3.1.5 Communication with test SP backend ... 17
3.1.6 Security ... 18
3.2 Android App Example ... 19
3.2.1 Getting up and running with the app example ... 19
3.2.2 Code Structure and Implementation overview ... 20
3.2.3 Webview HTML ... 21
3.2.4 Remember User ID ... 21
3.2.5 External Link Handling ... 21
3.2.6 Handling Long Click ... 21
3.2.7 Printing ... 21
3.2.8 Keyboard Handling ... 22
3.2.9 Logging ... 23
3.2.10 Cookies, Local Storage and Caching ... 23
3.2.11 Limitations on Android ... 23
3.2.12 Security ... 23
3.3 iOS App Example ... 25
3.3.1 Getting up and running with the app example ... 25
3.3.2 Code Structure and Implementation overview ... 25
3.3.3 Webview HTML ... 26
3.3.4 Web views ... 26
3.3.5 Remember User ID ... 27
3.3.6 Client presentation on iOS ... 27
3.3.7 External link handling ... 27
3.3.8 Handling long press ... 27
3.3.9 Printing ... 27
3.3.10 Cookies, Local Storage and Caching ... 28
3.3.11 Limitations on iOS ... 28
3.3.12 Security ... 29

References **32**

1 Introduction

In this document it is explained how to integrate the NemID JavaScript (JS) Client into a mobile app on the platforms iOS and Android. The documentation contains:

- Implementation guides
- Documentation of code example packages for iOS and Android

Initially a general explanation details the flow of events involved when NemID JS Client is integrated in an app, and therefore gives an overview of the implementation task that the service provider is faced with when integrating NemID JS Client into a mobile app. Furthermore, functionality which is general for all three platforms is detailed here. Following this, the contents of the code example packages are described, with this is included a short guide explaining how to get up and running with the two example apps (iOS/Android) that are associated with this document (**[Mobile-source])**. Functionality of the apps is then described and explained, and platform specific pointers are given.

1.1 Document Scope

This document serves as support for developers and architects, who wish to integrate the NemID JS client in to a mobile application (either iOS or Android). As such the focus is on the integration of the NemID JS Client and the functionality related to this in a mobile app context, and not to the same extent on the general functionality of the NemID JS client itself. For this reason, and to implement the NemID JS Client, the reader should be familiar with how the NemID JS client works in general. The documents "Implementation Guidelines for NemID (BANKS)" (**[Integration-Bank]**) and "Implementation Guidelines for NemID (OCES)" (**[Integration-OCES]**) describe the integration of the NemID JS Client for browser applications for Banks and OCES service providers (non-bank service providers) respectively.

2 NemID JS Client integration into Mobile Applications

This section gives an overview of how NemID JS Client should be integrated in a mobile application. Figure 1 in this document is closely related to the flow diagram in the Client Integration section of **[Integration-Bank]** and **[Integration-OCES]** (describing the NemID JS Client integration flow in general), and describes the overall flow in the case where NemID JS Client is integrated in a mobile application.

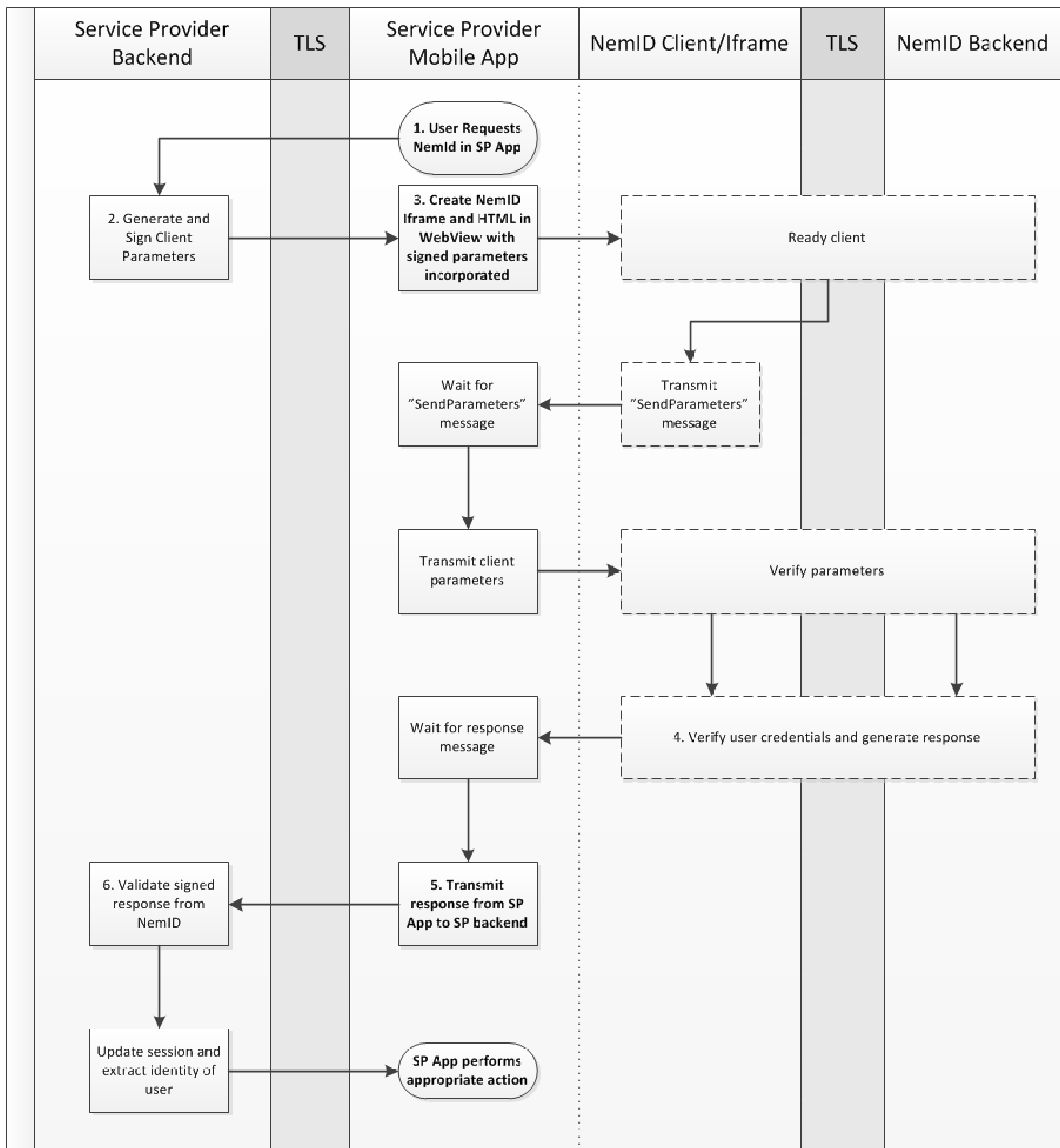


Figure 1 **NemID JS Client flow when integrated in a mobile application. Bold faced text marks elements that are different from the standard integration of NemID JS Client in [Integration-Banks]/[Integration-OCES]. Functionality encased in dashed boxes signifies functionality, which comes with the NemID JS client, and needs no implementation actions from the service provider.**

The NemID JS Client can be initialized and utilized as follows in a mobile application:

1. Upon request from the user of the mobile application the NemID JS Client is initiated
2. In order to start up the NemID JS client, signed parameters are required. These must be generated and signed on the service providers own backend (SP backend), and retrieved via a TLS

connection.

3. Upon retrieval of the signed parameters, these can be inserted into an HTML page loaded in a webview, through which the NemID JS Client can now be loaded. When the NemID JS client is loaded, it will request parameters from the HTML page via the "SendParameters" message.
4. When the NemID JS client is loaded, the user interacts with it.
5. The NemID JS client communicates with the NemID Backend, and upon response from the NemID JS client, the service provider app can retrieve the result of the authentication/signing from the webview and following this securely transmit the authentication/signing response to the SP backend.
6. The response must then be validated on SP backend. Given the result of the validation of the response, the service provider can now communicate to the service provider app which appropriate action to perform.

The above steps are elaborated on in the following subsections.

2.1 Parameter Generation

As stated above, before running the NemID JS client, it is necessary to obtain the signed initialization parameters from the service provider backend.

Retrieval of initialization parameters (for banks this includes the SAML request) must be conducted over a TLS connection ensuring that data is encrypted when transmitted between server and device[1].

The private key of the service provider is not secure on the device, for this reason, signatures and parameters should **not** be generated on the mobile device. **It is not recommended to generate any parameters on the device.**

For information on how to generate initialization parameters, please refer to the Client Integration sections of in **[Integration-Bank]** (banks) or **[Integration-OCES]** (non-bank service providers). For banks information on the generation of the SAML request can be found in the SAML Authentication section of **[Integration-Bank]**.

2.2 Response Handling

The app integrating the NemID JS client should take care of communicating the login/signing response received through NemID JS client from the NemID backend to the service provider backend. The service provider backend **must** then validate and decrypt the received response. The response should not be deemed valid before the service provider backend has validated the response with its private key. For details on how to perform validation of responses, please refer to the relevant integration documentation (**[Integration-Bank]/[Integration-OCES]**). The communication between app and service provider backend should be conducted over an encrypted (TLS) connection.

---

[1] Note that in signing flows the signtext is among the initialization parameters.

For a list of error codes, which can be received as responses, please refer to **[SP-Docs]** or https://appletk.danid.dk/developers/listerrorcodes.jsp (note that you must be a whitelisted developer to access this site, see more at **[Become-SP]**). It is the responsibility of the Service provider to implement appropriate error handling given these error-codes.

2.3 Iframe Integration and Running the Client

This section covers general settings needed for integration of the NemID JS client on mobile devices. For general information on integration of the NemID JS client into an iframe on a webpage (as well as in a webview) please see the Client Integration section of the relevant integration document (**[Integration-Banks]/[Integration-OCES]**). For details specific to the Android and iOS platforms respectively, please refer to the below designated sections 3.3, 3.3.12 and for iOS and Android see also the provided app examples, which are included in the code example package (**[Mobile-source]**).

2.3.1 Limited Mode and Client Size

NemID provides a responsive GUI mode called "limited" mode, which is specifically targeted for mobile applications. To employ this mode, the following must be done (more can be found about "limited" mode in **[Integration-Bank]** and **[Integration-OCES]** in the sections on UI Modes).:

- The URL used for launching the NemID JS client must be set as specified in the document "Configuration and Setup" (**[Config-doc])** among the documents in the NemID Service Provider package (**[SP-docs]**), i.e. with [mode]= lmt

When running in "limited" mode the NemID JS Client will adjust to the size of the iframe provided for it. In case of a very small sized iframe (smaller than 300x325px), for e.g. phones with small screens, the GUI will adapt to use a smaller screen with less text, designed for exactly this situation, otherwise the NemID JS client will run in a normal mode (both screen types are exemplified in the iOS and Android example apps within the code examples).

It is recommended, when sizing the NemID JS client in the mobile application, to do the following:

- On a phone:
  - Let the NemID JS client fill the screen of the phone, meaning that it is recommended to employ a dedicated webview in which the iframe is set to fill the screen (see examples below in Figure 2 and Figure 3).
- On a tablet:
  - For signing flows: it is recommended to let the NemID JS client fill the screen (see example in Figure 4).
  - For login flows: it is not recommended to let the NemID JS client fill the screen but to set the size by using an iframe with the width and height 320x460px (see example in Figure 5).

In general, it is not recommended that the minimum size of the iframe is smaller than 320x460px.

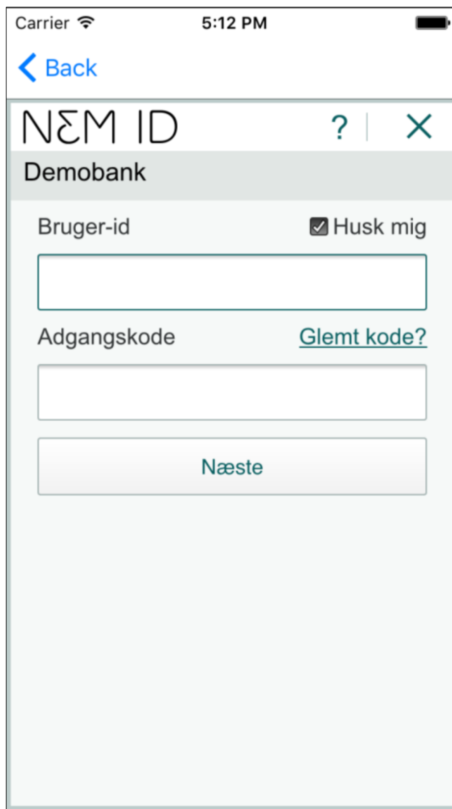


Figure 3 NemID Login, iPhone 6S simulator screenshot

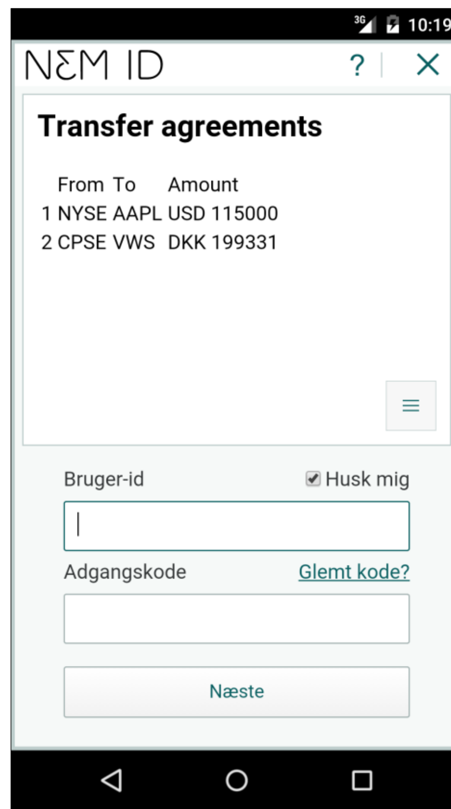


Figure 2 NemID XML Signing, Nexus 5 (Android phone) simulator screenshot

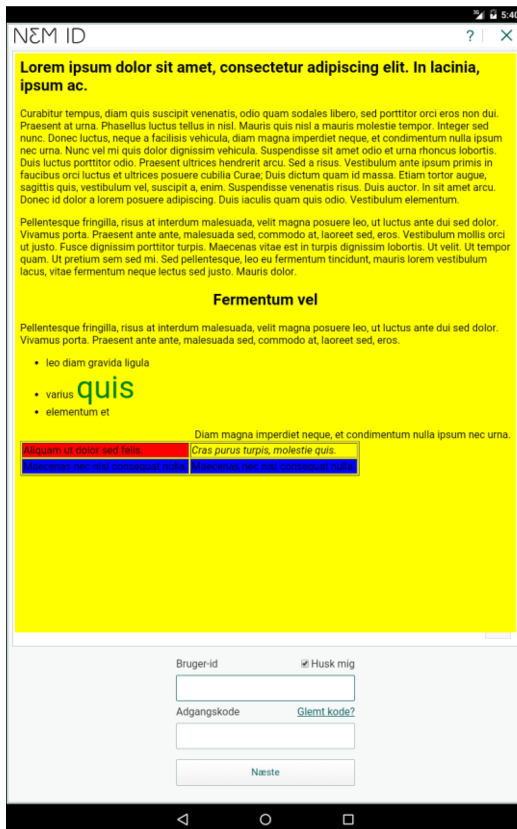


Figure 4 NemID HTML signing, Nexus 10 (Android tablet) simulator

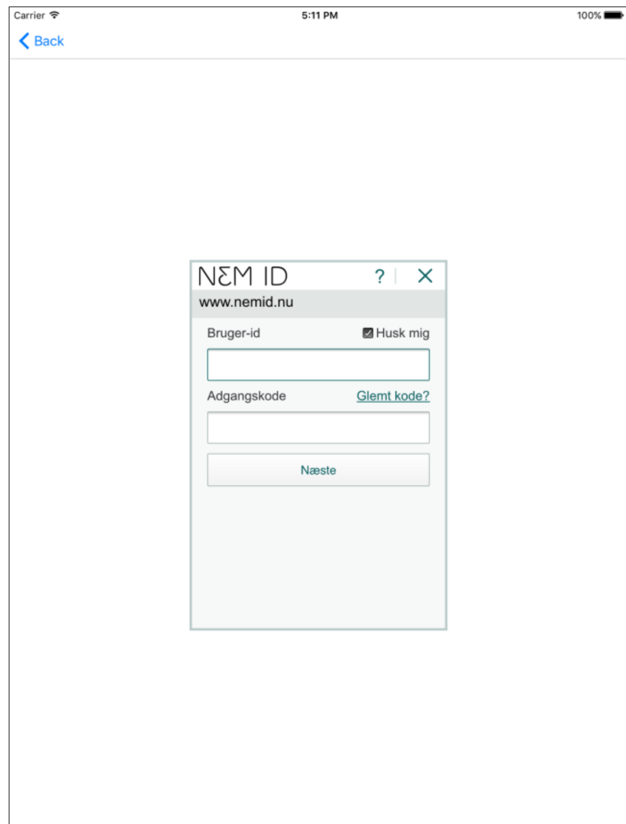


Figure 5 NemID Login, iPad Air simulator screenshot

Note that it is also possible to see examples of the GUI resized to other sizes on the developer site (https://appletk.danid.dk/developers), using the Custom Parameters Demo Login.

2.4 Integration of Remember User ID

If the user checks the "Remember me" check box above the user name, the Remember User ID functionality enables the username and password type to be remembered between app usages and is as such essential in order to facilitate ease of use of NemID JS, when integrated into a mobile application.

To enable this feature the use and storage of the Remember User ID token must be implemented in the app. The Remember User ID token, which can be retrieved from the signed flow response from NemID must be set if present in the flow response. This should then be retained in the app memory such that between app life cycles the token is stored, and that after re-starting the app, the token can be sent to the service provider to be incorporated in the signed parameters. **Note that it is highly recommended to implement the use of the Remember User ID functionality.**

This functionality is explained further in section 3.1.3.2 of this document and in the integration documents (**[Integration-Bank]**/**[Integration-OCES]**).

2.5 Integration of App Switch

The NemID JavaScript Client supports signalling that an app switch is ready to be performed. This will enable the integrating app to switch to the Common App that implements the NemID SDK that can perform the second factor authentication.

To enable the App Switch event, use the switch / checkbox under Advanced Parameters and perform a 2-factor flow. This will include `enableAwaitingAppApprovalEvent` in the startup parameters for the JavaScript Client. When getting to the second factor authentication, the JavaScript Client will now send the `AwaitingAppApproval` event, which is caught in the `NemIDViewController / NemIDActivity`.

The integration in the JavaScript Client is explained further in the integration document (**[Integration-Bank])**.

3 Android and iOS Source Example Packages

The documentation for integrating NemID JS into iOS and Android mobile applications comes with an accompanying code example package **[Mobile-source]**, consisting of the following:

- Demo service provider app examples for:
  - iOS
  - Android

The example apps serve as assistance in integrating the NemID JS Client in a mobile application.

The NemID JS client is supported on browsers on iOS and Android, for the specific support-coverage on mobile platforms please see the support-matrix at **[Technical Requirements]**.

3.1 General Functionality of the Example Apps

The two provided implementation examples show how to implement the necessary functionality to use NemID for authentication and signing on iOS and Android.

3.1.1 Supported Flows

The example apps support and exemplify how to implement the following NemID flows:

- Bank
  - These flows are relevant for banks only and are described in **[Integration-Bank]**. The following flows are available in the apps:
    - 2 factor bank login
    - 1 factor bank login
    - 2 factor bank signing
    - 1 factor bank signing

- OCES
    - These flows are relevant for non-bank service providers and are described in **[Integration-OCES].** The following flows are implemented in the apps:
        - 2 factor OCES login
        - 2 factor OCES signing

The following signtext-formats are implemented in the example apps for all platforms:

- TXT (plaintext)
- HTML
- XML
- PDF[2]

3.1.2 Functionality Overview

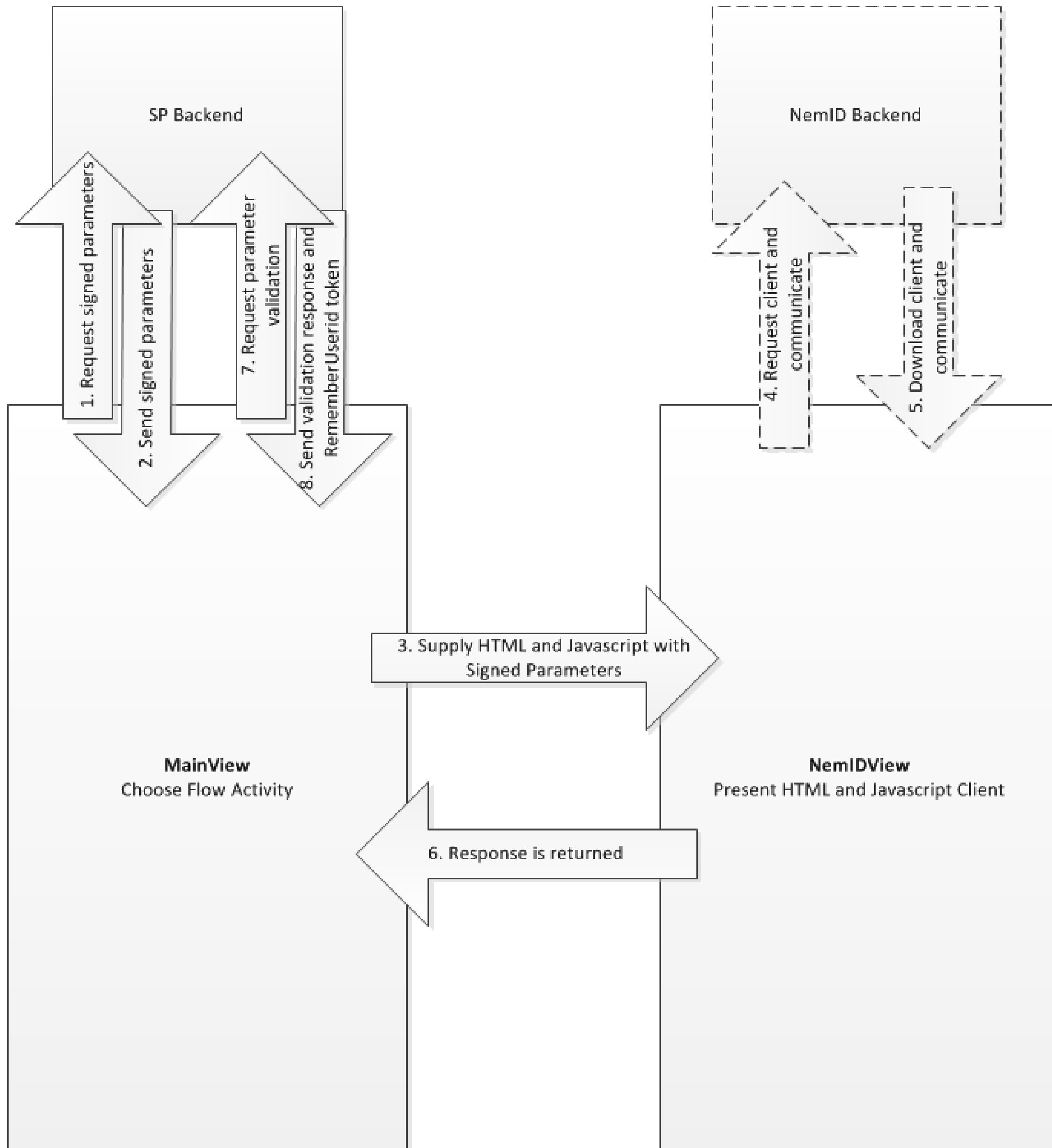
Each application is built up of two pages or views. The first view (MainView in Figure 6) controls, via a simple GUI, which NemID flow to run, while the second (NemIDView in Figure 6) is a webview holding or presenting an HTML page, runs the actual NemID JS Client. In the apps the MainView described here maps to the classes `MainActivity` (Android) and `MainViewController` (iOS) respectively. NemIDView maps to `NemIDActivity` (Android) and `NemIDViewController` (iOS).

Figure 6 describes how the app functions and essentially describes how the flow from Figure 6 is implemented in the app examples.

---

[2] PDF-signing is implemented here with the use of the whole (base 64 encoded) signtext, not with the use of external links (**[Integration-Bank]** and **[Integration-OCES]** describe the implementation of this).

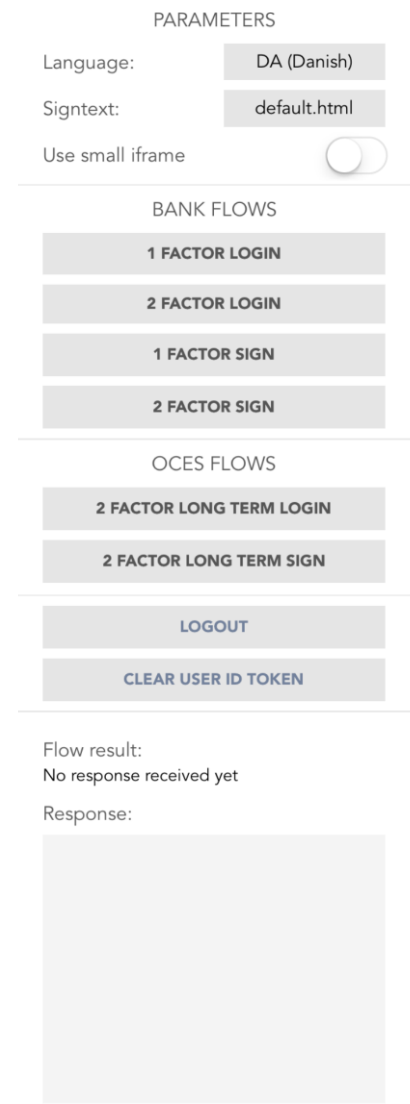
Figure 6 Example App, functional overview



3.1.3 MainView Options and App/Backend functionality

The MainView of each of the apps presents a simple GUI. Here it is explained in short how the apps function. Figure 7 shows the MainView from the iOS example app (`MainViewController`), the corresponding GUI in Android (`MainActivity`) is almost identical.

Figure 7 MainView screen-shot, showing Parameters, Flow Controls and Result section



The app GUI is divided in to four areas:

- Parameters
- Flow Controls
- Result
- Advanced Parameters

3.1.3.1 Parameters

This section lets the user control the most common parameters and settings needed.

Language

There is a language-picker, which enables the user to pick between a NemID JS client in Danish, English or Greenlandic.

Use small iframe

The example apps provide the possibility of forcing the small screen on devices which otherwise have a large enough screen to support the normal screens. This can be enabled by checking the "Use small iframe" checkbox.

Signtext-format

This dropdown lets the user pick which of the four formats to sign when running a signing flow: plaintext, XML, HTML or PDF.

3.1.3.2 Flow Controls

For each of the flows there is an activation button, which can start up the specific flow.

The flows are divided into two main categories Bank and OCES (see above Section 3.1.1 for explanation and description of supported flow types).

When initiating a signing flow, the user must first from the signtext-picker choose the desired signtext format, and then start the flow by pressing the relevant signing flow activation button.

As a default both the SP backend and the NemID backend is set to point to the NemID test environment (KOPI), for OCES, i.e. non-bank service providers this environment is also known as PP: https://appletk.danid.dk. On KOPI an example service provider backend is deployed. The example apps can request example parameters from this backend for example usage.

KOPI has an implementation of both the NemID backend and a test service provider backend for mobile app usage (parameter generation and response validation). In order to best demonstrate the functionality, as a default the app therefore points to KOPI for both backends.

In the code base of each of the apps there is however a clear logical separation of the two backends (and their URLs). This enables the possibility of swapping in the service provider's own backend for parameter generation easily. See more about the communication with the SP backend in the example apps in Section 3.1.5.

Logout and 1 factor signing (Only relevant for banks)

The Logout button is only relevant for bank flows as it clears the Hsession token if this is saved on the backend session on the test SP backend on KOPI. Furthermore, the ALLOW_STEPUP parameter is implemented as the default. For more on these topics, see **[Integration-Bank].**

Clearing User ID token and the Remember User ID functionality

NemID contains a functionality called Remember User ID, this functionality enables NemID to be started up with the UserId of the user filled in, and with the 4-digit password mode GUI, if the user has a 4-digit password (see sections on Remember User ID and the Encrypted Assertion in **[Integration-Bank]** and section on Remember User ID in **[Integration-OCES]**). It is essential to implement the Remember User ID functionality in mobile applications, as this enables the NemID JS client to be loaded with the 4-digit password GUI, when the user has such a password.

Any NemID JS client parameter response from the test SP backend on KOPI will include the REMEMBER_USERID parameter. This is the recommended practice when utilizing NemID in a mobile application.

If no value for this parameter is provided in the request to the test SP backend on KOPI, the response will contain the parameter with the value empty ("REMEMBER_USERID":"").

When the REMEMBER_USERID parameter is empty, the user is presented with a check box in the NemID JS client, with the text "Remember Me", which enables remembering the user. If the user checks this box, upon a successful response this response will include a REMEMBER_USERID token value.

This token must be retrieved by the SP backend and communicated back to the app for storage (which is also implemented in the example apps and on KOPI). The token is included in the encrypted assertion in bank flow responses, and in the signed assertion for an OCES response.

If the REMEMBER_USERID token is included in the request to the test SP backend on KOPI, then the REMEMBER_USERID parameter will have this value filled in in the signed NemID JS client parameters.

The Clear User ID button clears the REMEMBER_USERID from app storage if this is stored in the app.

For a more detailed explanation of the use of REMEMBER_USERID please refer to **([Integration-Bank]**/**[Integration-OCES]**). Here information can be found on how to retrieve the remember user id token from the response received from the NemID JS Client.

3.1.3.3 Result

This section holds the results of a NemID flow. It is divided into two parts a Status field and a Response text field.

Response Text Field

In the bottom of the GUI a large text field is reserved for holding the response received from the NemID backend, this is shown after a flow has completed (either successfully or with errors).

Flow Result Field

Above the response text field, there is a status code field, this will either state that the flow succeeded, failed or was not valid based on the response received and validation of the response by the service provider backend.

3.1.3.4 Advanced Parameters

In the bottom of the MainView a section with more advanced parameters is given (not shown in Figure 7). In this section it is possible to adjust the 1 factor signing mode, give specific values to the iframe holding the NemID JS client and to specify another backend URL (this will however by default point to the same backend for service provider and NemID). These features are only useful for more specialized testing. Initially the parameters from the above section should be sufficient for understanding, testing and implementing all common use of the NemID JS client in a mobile app.

You can furthermore enable the use of App Switch and suppressing push to a device. Enabling App Switch will allow the integrating app to switch to another app that can perform the second factor authentication, as discussed in 2.5. Suppressing push to a device will disable sending a push notification through Firebase and only allow the current transaction to be pulled.

3.1.4 General App Limitations

The example apps provided in this documentation package are meant as examples and as such are not production ready and should not be run in their current form in a production environment or against the NemID Production backend.

3.1.4.1 Logging

Care has not been taken with regard to logging in the example apps, in fact logging has been added specifically for instructional purposes, and only to be used in test environments.

It is the responsibility of the service provider that no sensitive material is logged in a production environment.

3.1.5 Communication with test SP backend

As mentioned earlier, a "dummy" backend is available on KOPI (https://appletk.danid.dk). The source code for this backend is not supplied as part of the example packages. For implementation of backend functionality, please refer to **[SP-docs]**.

The backend deployed on KOPI is able to supply signed test parameters to start flows against the NemID backend on KOPI. Furthermore, it can validate SAML (bank login results) and XMLDSIG (all signing results) responses signed by the NemID backend on KOPI as the result of a successful flow. Examples of the use of this backend are incorporated in the app examples.

As described in Figure 6, when starting a flow, the app retrieves signed parameters from the test SP backend on KOPI. After this the parameters are inserted into an HTML page, which is rendered in a webview. The user then completes a flow. The result of this flow is retrieved by the app from the HTML page. Rudimentary error handling is employed at this point. The app will attempt to validate any response, if the response could not be validated by the backend, then this is communicated to the app, and this is stated in the "Flow result" text-box of the MainView. The flow response is posted in the "Response" text-box of the MainView. This means that if there for some reason is a flow error, example could be that the digest of the initialisation parameters is incorrect (causing an APP001 error), the "Flow result" will state: "Could not validate response." and the "Response" will state the APP001 error response. In case the flow could be validated by the test SP backend the "Flow result" will state: "Flow success. Response valid." and the "Response" will show the response received from the NemID JS client.

3.1.5.1 Limitations of test SP backend

It must be stressed that the communication protocol and the implementation of this between app and test SP backend used in the app examples is merely a rudimentary example showing the intended functionality. The service provider must ensure to employ appropriate error handling. Furthermore, as mentioned above, this communication must be conducted over TLS. For further information on response handling see section 2.2. Finally, in the test apps, synchronous network communication is employed in some places, it is the responsibility of the service provider to evaluate whether this is appropriate for the application and implement corresponding asynchronous calls and handling if necessary.

3.1.6 Security

The app example contains some examples of best practice security measures which are recommended to include. This section describes these general measures. Note that these measures are not thought to be exhaustive in terms of securing or hardening an app. Furthermore, the absence of the use of some of the measures does not necessarily make the app insecure – these are simply some best practices to take in to account when using the app examples in building your own application.

Each of the platform specific details of these measures are detailed further in the following sections on the Android and iOS example. Several of these tips are also featured in the OWASP Mobile Security Testing Guide (MSTG)[3].

Do not distribute the parameter signing key in the app, store it on the backend

The private key used for signing parameters is not secure on the device, store it in the backend, and procure signed parameters from the backend via the app.

Use TLS (Transport Layer Security) for all network connections

Whenever loading any content via the network, be this from the backend, when procuring signed parameters or when communicating the login or signing response to the backend, ensure to use TLS. For the purposes of the Webview, this means communicating over HTTPS. The NemID JavaScript client cannot be loaded via HTTP, if the HTML for loading the iframe for loading NemID JavaScript is loaded from a backend, this must be done via TLS, see more in the next section on Hardcoded HTML. Using TLS protects the confidentiality of the data being transferred and protects it against tampering. The authenticity of the backend is also ensured, meaning that you can count on the server that you are communicating with is the correct one (this measure can be strengthened using the concept of Certificate Pinning, we do not cover this here, but recommend using Certificate Pinning when possible).

Each of the platforms offer different ways of enforcing that TLS is used, see the Security sections for each of the app examples, see 3.2.12 (Android) and 3.3.12 (iOS) for more on this.

Use Hardcoded HTML for loading the iframe (avoid dynamic loading when possible)

Each of the app examples use hardcoded HTML and JavaScript to load the iframe which loads NemID JavaScript, this reduces the amount of dynamic content to be loaded in the Webview. Evaluate whether this is a good approach for your app, since while there a security benefits in reducing the amount of dynamic content, it may be more flexible to load the site from a backend. If possible, the content should just be kept static. If the HTML is loaded from a file, restrict the files that can be loaded locally in the Webview.

Remove unused functionality

The app examples include print functionality, in the case where only NemID login functionality is used in an app, there is no reason to include this functionality. Best practice is to simply remove this functionality if it is not necessary.

---

[3] https://www.owasp.org/index.php/OWASP_Mobile_Security_Testing_Guide

Remove log statements

Log statements such as console.log, NSLog and Android Log statements should be removed from any production source.

Do Origin-checking of post-message from the NemID JavaScript iframe

It is recommended to do origin-checking on the messages received from the iframe. This is implemented in the HTML used in the app-examples. The messages from the iframe should always originate from the applet.danid.dk domain in Production and appletk.danid.dk domain in the NemID test environment (KOPI).

Restrict unnecessary file access from the Webview

As stated above, if HTML is loaded from a file stored in the app, it is recommended to restrict the files which can be loaded in the app to this file. If this is not the case, it is recommended to restrict file access from the Webview completely. See more on this in the Security sections for each of the app examples, see 3.2.12 (Android) and 3.3.12 (iOS).

Avoid supporting or only use hardcoded HTML for Android versions lower than 4.2

Webviews on Android before version 4.2 have a vulnerability which, given the injection of malicious JavaScript to the page being loaded in the Webview, can expose data in the native app via reflection[4].

Hardcoding the HTML reduces the possibility of injecting JavaScript to the Webview. Avoiding the support of lower versions than 4.2 (API level 17), ensures that this is not an issue. Note that the issue is also present on higher versions of Android if the app is built with a minimum target at API level 16 or lower.

3.2 Android App Example

This section describes the contents of the Android Example App Source package and how to setup a running example app.

3.2.1 Getting up and running with the app example

It is recommended to run the Android app example using the Android Studio IDE (http://developer.android.com/develop/index.html). Android Studio can be used to run the example in a simulator on the desktop. A wide range of simulators are available for this purpose, enabling testing in all Android versions. From this IDE the app example can also be deployed to a physical Android device. If running using Gradle, remember to update Gradle and synchronize project with Gradle files.

[4] https://labs.mwrinfosecurity.com/blog/webview-addjavascriptinterface-remote-code-execution/

3.2.2 Code Structure and Implementation overview

The MainView (see Figure 6 and Figure 7) functionality is implemented in `MainActivity`. From `MainActivity` the flow options are set up, and the chosen flow is started by using the NemIDView (see Figure 6) implemented in `NemIDActivity`.

The following warnings have been supressed in the demo project:

- Lint warnings enabling JavaScript ("`SetJavaScriptEnabled`", "`AddJavaScriptInterface`") since these are required
- "`NewAPI`" to suppress API usages after `minSdkVersion`.
- "`deprecation`" to suppress warnings about using older APIs.

3.2.2.1 Entry point code and short introduction to implementation

This section summarizes the implementation in short highlighting the important parts of the code.

`MainActivity`

When the user presses a flow button in the MainView, the method `startFlow` will be called with the set parameters, as such this is the entry method in the Android example to starting up a NemID flow.

The `startFlow` method in short does the following: the SP backend is called to retrieve signed parameters. These are set in a global variable on the `MainActivity` class (this is intentional, since these often hold too much data to incorporate directly in an intent). Clientsize and NemID backend url is sent on an intent, which is used to start up the `NemIDActivity` (NemIDView).

`NemIDActivity`

In the `onCreate()` method of `NemIDActivity` the following happens to load the NemID JS Client in a webview: The flow specific settings are read from the received intent. Following this the webview is created. The webview created is an instance of the class `NemIDWebView`, which incorporates, together with `NemIDActivity`, special soft keyboard handling necessary to ensure a good user experience when using the NemID JS Client (read more on this below in 3.2.8). Relevant settings are then set on the webview. The signed NemID parameters (which are retrieved from a public variable on the `MainActivity`) for the html page are incorporated into the html, which is then passed on for loading by the webview.

It is recommended to incorporate the `NemIDActivity` and `NemIDWebview` (and classes that these rely on) in the service provider application as is, as these incorporate several parts, which are necessary for the NemID JS Client to function optimally in an Android application, such as: customized keyboard handling, printing capabilities and long click handling. All of these parts are described below.

3.2.3 Webview HTML

The html to be loaded by the webview is retrieved for loading using the `getHtml` method. This html incorporates JavaScript (retrieved via the method `getJavascriptSrc`), which implements all communication between webview and App. The NemID JS Client communicates with the html page via messages (as described in the **[Integration-Bank]**/**[Integration-OCES]** documents, see Client Integration sections. Webview to App communication works via the custom JavaScript interfaces defined in `NemIDActivity` and `NemIDWebView`, the most important example being the action when the "`changeResponseAndSubmit`" message is received, in this case the NemID JS client serves a flowresponse to the HTML, which is then communicated via the `getResponse()` method in the JavaScript interface defined in `NemIDActivity`.

3.2.4 Remember User ID

On Android the Remember User ID token is saved to app memory using `SharedPreferences`, this ensures that the token is stored across app life cycles. This is done using `setRememberUseridToken` in `MainActivity`.

3.2.5 External Link Handling

External links can be found embedded in the NemID JS client (help pages, forgotten password etc.) and in sign texts. External links must be opened in either a new web view inside the app or by opening a browser app. Filtering of "safe" links is not recommended since it could stop internal links in the NemID JS client from showing.

3.2.6 Handling Long Click

Since the NemID JS client is embedded in a web view, it is by default allowed to long click and select elements. However, this can be confusing for the mobile user, since buttons and graphics are not usually selectable in a mobile application. The demo app shows how to disable long press for Android webviews (`setOnLongClickListener` in the `onCreate` method in `NemIDActivity`).

3.2.7 Printing

It is possible to print signtexts via the print icon in the signing flows. Programatically printing from the app should be triggered when "`RequestPrint`" is received in JavaScript of the HTML of the webview. The

print content is downloaded as HTML. In the demo app printing is handled in the `requestPrint` method in `NemIDActivity`. There are limitations to printing on older Android versions and currently only printing of plaintext, HTML and XML type signtexts is possible (see limitations sections below).

3.2.8 Keyboard Handling

Due to default soft keyboard handling in Android and the many different manufacturers of Android devices employing different soft keyboard handling, customizations have been implemented in the code examples to secure a good user experience, when using NemID in an Android application. These customizations should be employed when integrating NemID JS in an Android application. The customizations are described in detail below. There are however still some limitations (see Limitations on Android below).

Ensuring keyboard availability

By default, for usability in the normal case, the keyboard does not pop up on webpage in an Android browser, when focus is set in a field. The following customization is employed in the example app: When the NemID JS client is loaded the message "`RequestKeyboard`" is received in JavaScript from the NemID JS client. Using this, the soft keyboard can be triggered via the JS interface in `NemIDActivity`.

Ensuring correct keyboard layout

When the NemID JS Client is loaded, the keyboard should be popped up with the relevant layout (numerical/alpha-numeric). Some Android devices will always show the alphanumeric keyboard in this case (even though a numeric input type is defined on the input field). To ensure that the numeric keyboard is shown, when a user encounters the 4-digit password boxes, further handling must be added for this. Handling is implemented in the app example package: The NemID JS Client will communicate which input type is requested for the field in question, when sending the "`RequestKeyboard`" message. The `NemIDWebView` defines a custom JavaScript interface, with the method `setKeyboardType`, which is used to set the keyboardtype.

Handling Backspace in the 4-digit case

On older Android versions (< API level 22), it may not be possible to delete previous password characters, unless specific handling is implemented. The problem arises since the `KEYCODE_DEL` (backspace) keyevent is not always handled correctly when employed in a webview. A customized fix has been implemented, please see implementation for details.

Adjusting GUI to account for Keyboard pop-up

In order to ensure that the keyboard does not cover the input fields, the `ViewTreeObserver.OnGlobalLayoutListener` to decide when to adjust the GUI as this is not done automatically on all Android devices.

3.2.9 Logging

The JavaScript console.log statements are written to the Android log, so this information is available for debugging.

3.2.10 Cookies, Local Storage and Caching

Cookies, local storage and caching are enabled on the webview. These should not be disabled as they benefit performance of the NemID JS Client.

3.2.11 Limitations on Android

- **Printing**: Android printing was first introduced in API 19 (will not work for API 16,17,18). Printing of PDF signtexts is not currently supported.

- **Keyboard layout on HTC devices running older Android versions:** It has been found that on some HTC devices running <API level 22, keyboard is not restarted upon focus to a new input field, meaning that showing the phone keyboard (numeric only) may leave the user stuck with the numeric keyboard, meaning that the user is not able to type in alphanumeric characters if this is necessary. For this reason, the alphanumeric keyboard is forced on these devices.

- **Rotation:** Width and height are set automatically to a fixed number of pixels upon load of `MainActivity` depending on device size and are not changed upon rotation. Changing the sizes on rotation or setting sizes in percentages currently gives unpredictable behaviour on some devices. On some phones and smaller tablets rotating with fixed width and height may yield undesirable layout. For this reason, the recommendation is to do as in the app example, where the app is locked in portrait mode for devices with configurations other than `Configuration.SCREENLAYOUT_SIZE_XLARGE` (large tablet size).
  Furthermore, to avoid reloading the contents of the webview upon rotation (since it will break the NemID flow), the following has been added to the Android manifest:
  `android:configChanges=”keyboard|keyboardHidden|orientation|screenSize”`

3.2.12 Security

This section describes some of the platform specific security measures or best practices included in the app example for Android, and some possible additions. For a more general description, please see section 3.1.6.

Restrict unnecessary file access from the Webview

Since loading of local files and resources is not necessary in the app example, these features are disabled.

```
// Disabling access to files and other content
settings.setAllowFileAccess(false);
settings.setAllowFileAccessFromFileURLs(false);
settings.setAllowUniversalAccessFromFileURLs(false);
settings.setAllowContentAccess(false);
```

Enforcing TLS

It is recommended to set the Network Security configuration to disallow cleartext traffic:

```
<network-security-config>
    <base-config cleartextTrafficPermitted="false">
    </base-config>
</network-security-config>
```

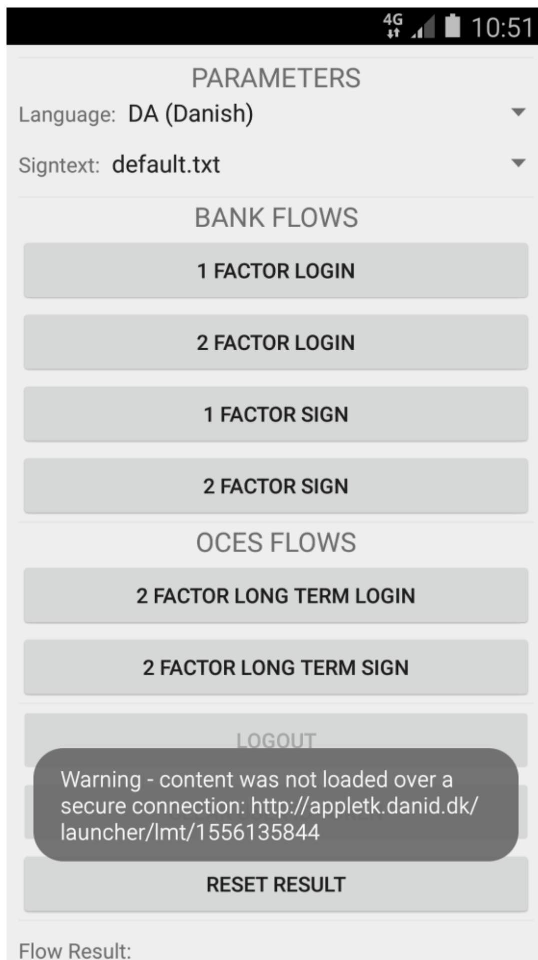
This is demonstrated in the app examples, where this configuration is referenced by the attribute android:networkSecurityConfig. The networkSecurityConfig attribute is ignored below API level 24, so to cover API level 23 the example apps also have the following Android manifest attribute:

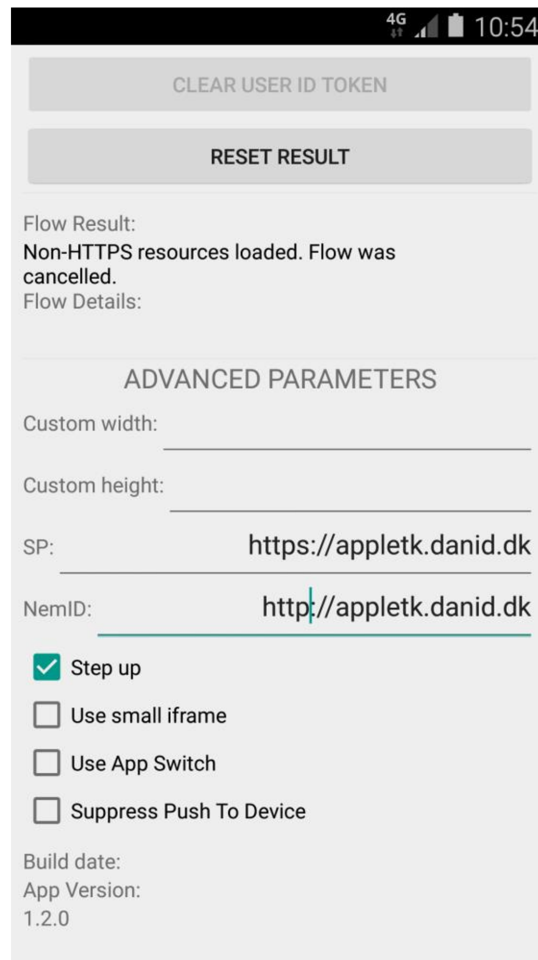
```
android:usesCleartextTraffic="false"
```

The usesCleartextTraffic attribute is ignored when attribute networkSecurityConfig is present. Below API level 23 there is no direct system support for blocking cleartext traffic and restrictions must be implemented manually. This is demonstrated in the app examples by overloading WebViewClient.shouldInterceptRequest to intercept all cleartext traffic from the webview. This mimics the hasOnlySecureContent method on webview in iOS, see section 3.3.12

According to the Android documentation these cleartext settings ensure that loading over the network is done via TLS on a best effort basis. Some third-party libraries may not honour this, and the `Webview` class does not honour this attribute for API levels lower than 26[5]. For these API levels it is encouraged to implement a manual check for non-TLS traffic. This is demonstrated by the mentioned overloading of shouldInterceptRequest, where a warning will be given, and the flow terminated if the webview loads a non-https resource:

5 https://developer.android.com/reference/android/security/NetworkSecurityPolicy.html#isCleartextTrafficPermitted()





Note that external URLs are loaded in a browser in the example apps, rather than in the webview, but they are also subject to a cleartext warning due to the overloading of shouldInterceptRequest.

3.3 iOS App Example

This section describes the contents of the iOS Example App Source package and how to setup a running example app.

3.3.1 Getting up and running with the app example

The demo project can be run by opening `TestNemIdJavascript.xcodeproj` in the iOS source library. The project requires Xcode 7.

3.3.2 Code Structure and Implementation overview

The MainView (see Figure 6 and Figure 7) functionality is implemented in `MainViewController`. From the `MainViewController` the flow options are set up and the chosen flow is started by pushing the `NemIDViewController` (NemIDView, see Figure 6), this loads the NemID JS Client in a webview via the method `loadNemID()`.

3.3.2.1Entry point code and short introduction to implementation

This section summarizes the implementation in short highlighting the important parts of the code.

`MainViewController`

When the user user pushes a flow initiating button in the MainView, the method `startFlow()` is called. This method takes care of retrieving signed NemID parameters from the SP backend, and sets variables (e.g. NemID JS client size, user specified settings etc.) needed for NemID JS client initialization on the `NemIDViewController`. Finally, it pushes the `NemIDViewController` to load NemID in a webview.

`NemIDViewController`

When the `NemIDViewController` is pushed and the lifecycle method `viewDidAppear` is called, the NemID JS client is loaded via the `loadNemID` method. `loadNemID` incorporates the values set by `MainViewControler` into the html and JavaScript, which is then loaded in the webview (among these are the signed NemID parameters).

It is recommended to incorporate the `NemIDViewController` as is, as this holds the functionality needed for integration of the NemID JS Client.

3.3.3 Webview HTML

The html to be loaded by the webview is defined in the `loadNemID` method of `NemIDViewController`. This html incorporates JavaScript (retrieved via `getJavascript`), which implements all communication between Webview and App. The NemID JS Client communicates with the html page via messages (as described in the **[Integration-Bank]**/**[Integration-OCES]** documents, see section on Client Integration, in the subsection on Start-up and handling responses). Webview to App communication works via the request intercept in `shouldStartLoadWithRequest`, which intercepts requests using a defined url scheme. Upon intercept the `stringByEvaluatingJavascriptFromString` is then used to call into the JavaScript of the webview to retrieve content.

3.3.4 Web views

On iOS there are two different web views, `UIWebView` and `WKWebView`. Due to better performance of JavaScript it is recommended to use `WKWebView`. `WKWebView` is a light weight version of `UIWebView` and has some limitations compared to `UIWebView`, so it is advised to investigate the differences before choosing a web view to make sure it meets the needs of your app.

Note: The standalone `SFSafariViewController` should NOT be used for the NemID JS client, as it has no way of communicating through JavaScript and cannot be verified as safe for NemID transactions.

3.3.5 Remember User ID

On iOS the Remember User ID token is saved to app memory using `NSUserDefaults`, this ensures that the token is stored across app life cycles. This is done using `setRememberUseridToken` in `MainViewController`.

3.3.6 Client presentation on iOS

Since iOS does not have a hardware back button, it is recommended to always have a navigation option to exit the NemID JS client flow. Using a full screen web view for the NemID JS client without a back button can leave the app in an unrecoverable state if e.g. the internet connection is lost. To demonstrate the recommended navigation, the demo app is made with a visible back button.

3.3.7 External link handling

External links can be found embedded in the NemID JS client (help pages, forgotten password etc.) and in sign texts. External links must be opened in either a new web view inside the app or by opening Mobile Safari or similar mobile browser app. Filtering of "safe" links is not recommended since it could stop internal links in the NemID JS client from showing.

3.3.8 Handling long press

Since the NemID JS client is embedded in a web view, it is by default allowed to long press and select elements. However, this can be confusing for the mobile user, since buttons and graphics are not usually selectable in a mobile application. The demo app shows one way to disable long press gestures for `UIWebView` (`'disableLongPressGestures'` in `NemIDViewController`). See Limitations section regarding disabling long press on `WKWebView`.

3.3.9 Printing

It is possible to print signtexts via the print icon in the signing flows. Programatically printing from the app should be triggered when "`RequestPrint`" is received in JavaScript of the HTML of the webview. The print content is downloaded as HTML. There is a demo implementation using the `UIPrintInteractionController` in `NemIDViewController`. Printing of PDF signtexts is currently not supported (see limitations below). Printing of all other signtexts: plaintext, XML and HTML is however supported.

3.3.10 Cookies, Local Storage and Caching

Cookies, local storage and caching is enabled on the webview by default. These should not be disabled as they benefit performance of the NemID JS Client.

3.3.11 Limitations on iOS

- **Long press** `WKWebView`: The demo app currently does not implement disabling of long press gestures on `WKWebView`. It is currently unknown if this is supported.
- **Keyboard handling**: A JavaScript message "`RequestKeyboard`" can be received when the keyboard should be displayed to the user. This is not implemented in the iOS demo app and currently it is unknown if this functionality can be implemented.
- **Focus in the username, but can't enter characters (UIWebview)**: Due to a bug in UIWebview, when keyboardDisplayRequiresUserAction is set NO, then the keyboard pops-up automatically with focus in the username, however, it is not possible to enter characters, and it only becomes possible to enter characters once another field has had focus, and focus is then brought back to the username field by the user. This bug is not present when using WKWebview. However, in one scenario, iOS chooses UIWebview on your behalf: When a webpage, using the meta-tag `<meta name="apple-mobile-web-app-capable" content="yes">`, is stored to the home screen, then the webpage is opened in a native app context, by iOS, using UIWebview with keyboardDisplayRequiresUserAction set NO, meaning that the issue is present in this case. One workaround is documented below in 3.3.11.1.
- **Printing PDF**: There is currently no support for printing PDF signtexts from the NemID JS client on iOS. However, it is always possible to make the document available inside the app using NemID.
- **Rotation:** Width and height are set automatically to a fixed number of pixels upon load of MainViewController depending on device size and are not changed upon rotation. Changing the sizes on rotation or setting sizes in percentages may give unpredictable behaviour on some devices. On phones rotating with fixed width and height may yield undesirable layout. For this reason the recommendation is to do as in the app example, where the app is locked in portrait mode for phones (this is done in the associated .plist configuration file).

3.3.11.1 Focus in the username – a workaround

Add a text field, that can't be seen, because it is inside a div with width and height 0. It should be placed right before the iframe to avoid too much scrolling on the page:

```
<div style="width: 0; height: 0; overflow: hidden;">
   <input type="text" id="tmpfocus" autocomplete="off" autocorrect="off"
autocapitalize="off" spellcheck="false" />
</div>
<div id="nemid_layer">
```

```
    <iframe...
```

You can't use style "display: none;" to start with, since it is not possible to set focus on a hidden field.

One further enhancement, to avoid scrolling, would be to set the position to be relative with some fitting "top"-value, so the invisible input field has the same vertical position as the input field in the JS client. For instance, like this:

```
<div style="width: 0; height: 0; overflow: hidden;">
    <input type="text" id="tmpfocus" autocomplete="off" autocorrect="off"
autocapitalize="off" spellcheck="false" style="position: relative; top: 113px" />
</div>
<div id="nemid_layer">
    <iframe...
```

In the code listening for events from the NemID iframe, set focus on the invisible field and set display none, when you get the "RequestKeyboard" event:

```
// This should be a page "global" JavaScript parameter.
var setTmpFocus = true;
...
    // This should be inside function receiving events from the NemID iframe.
    if (setTmpFocus && message.command === "RequestKeyboard") {
        var elm = document.getElementById('tmpfocus');
        elm.focus();
        elm.style.display = 'none';
        setTmpFocus = false;
    }
```

- The setTmpFocus variable is just to ensure the JavaScript is only executed once.
- style.display has to be set to ‘none’ after setting focus, since otherwise the caret in the text field will be visible when running the iOS homescreen "app", even if the text field is invisible.

With this workaround, the keyboard pops up, but the focus is removed from the user id field immediately, and when the user taps on the field, he will be able to input text.

Note: The "RequestKeyboard" event may not be fired in signing scenarios. Alternatively, it should be possible to create some JavaScript, that checks, when the iframe is done loading, but this is not shown here.

3.3.12 Security

This section describes some of the platform specific security measures or best practices included in the app example for iOS, and some possible additions. For a more general description, please see section 3.1.6.

Restrict unnecessary file access from the Webview

As opposed to Android, on iOS file and content loading restrictions are enabled on the Webview by default.

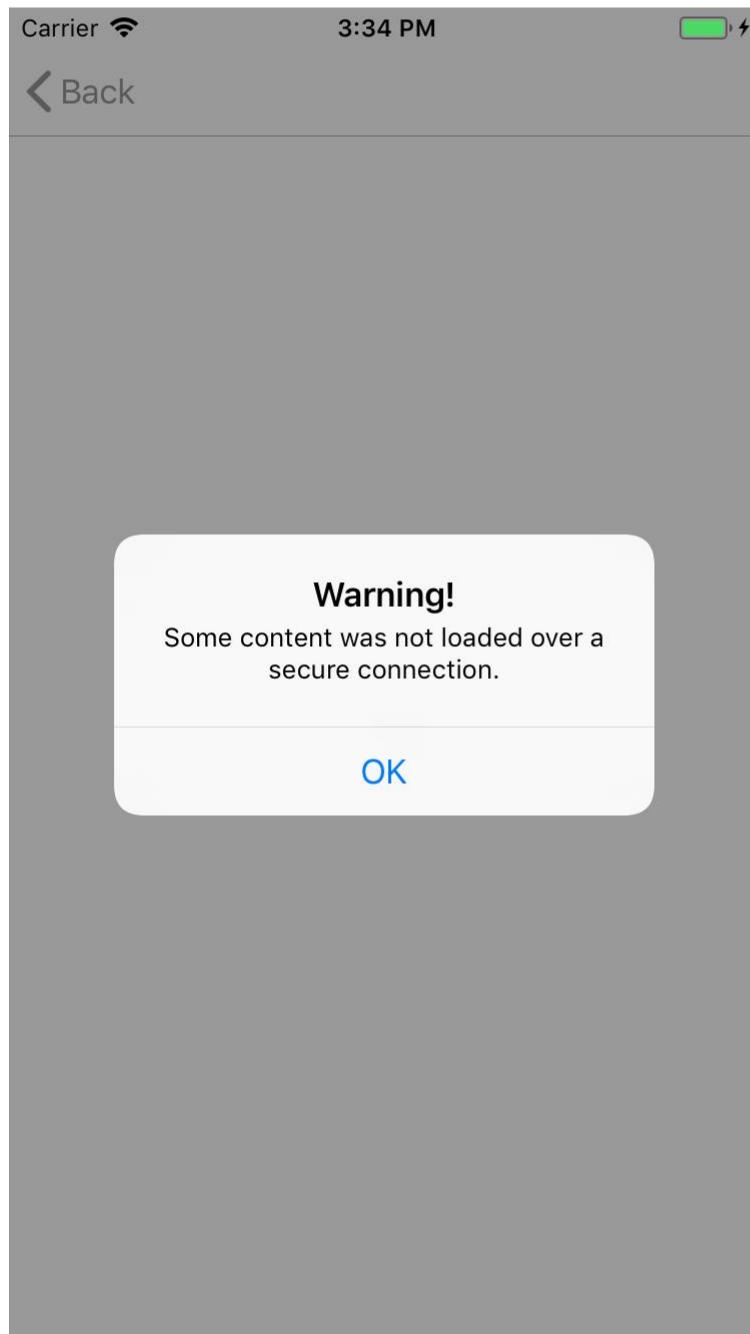
Enforcing TLS

With iOS 9, Apple has introduced App Transport Security (ATS), which is enabled by default. ATS enforces all connections to use HTTPS, using TLS 1.2 or higher, and using best practices for secure communication. ATS can be disabled for internal testing purposes, but it is strongly advised to keep ATS enabled in any publicly available app.

Note that ATS is only active when the address being loaded has a hostname, i.e. if the address being loaded is an IP-address, ATS is not active[6].

WKWebview provides the method hasOnlySecureContent, which can be used to check whether any loading of data over the network in the Webview is done without the use of TLS (even when using an IP address instead of a hostname address). This has been implemented in the example such that if any content loaded in the Webview is not loaded over TLS, a prompt is shown to the user as shown below and the flow is dismissed.

6 https://developer.apple.com/library/archive/documentation/General/Reference/InfoPlistKeyReference/Articles/CocoaKeys.html#//apple_ref/doc/uid/TP40009251-SW33


Carrier
3:34 PM
Back
Warning!
Some content was not loaded over a secure connection.
OK

References

Reference ID	Document Name
[Integration-Bank]	"Implementation Guidelines for NemID (BANKS)" NemID Integration - Bank.pdf Available in release package for banks.
[Integration-OCES]	"Implementation Guidelines for NemID (OCES)" http://www.nets.eu/dk-da/Service/kundeservice/nemid-tu/tjenesteudbyderpakkeJS/Documents/NemID%20Integration%20-%20OCES.pdf
[SP-Docs]	NemID Service Provider documentation package In Danish: http://nets.eu/tu-pakke In English: http://nets.eu/sp-package
[Config-doc]	"Configuration and Setup", v. 1.16 http://www.nets.eu/dk-da/Service/kundeservice/nemid-tu/NemID-tjenesteudbyderpakken-okt-2014/Documents/Configuration%20and%20setup.pdf
[Mobile-source]	For non-bank service providers, this is avalailable under "Kildekode": In Danish: http://nets.eu/tu-pakke In English: http://nets.eu/sp-package For banks: This is part of the release available for banks, package is named Mobile Example NemID Javascript.
[Become-SP]	Bestil NemID Tjeneste Udbyder: http://nets.eu/tu-bestil
[Technical Requirements]	https://www.nemid.nu/dk-da/om-nemid/tekniske_krav/