AusweisApp2 SDK

Release 1.16.2

Governikus GmbH & Co. KG

May 15, 2019

Contents

1 Introduction

This documentation will explain how to initialize and start up the AusweisApp2 as an additional service. It distinguishes between a connection to the application and the communication between your application and AusweisApp2.

The section *Connection* (page 2) will show you what you need to do to set up a connection to AusweisApp2. Once you have established a connection you can send and receive JSON documents in a bi-directional manner. There are different commands and messages. These are listed and described in the section *Protocol* (page 16). The protocol is split up in *Commands* (page 16) and *Messages* (page 21). Commands will be sent by your application to control AusweisApp2. Messages contain additional information to your command or will be sent as an event.

Also this documentation provides some example workflows to show a possible communication.

Important: The AusweisApp2 does **not** provide any personal data to your client application directly as AusweisApp2 does not have access to this data for security reasons. AusweisApp2 facilitates a secure connection between the eID server and the ID card, enabling the eID server to get those data from the card.

This way your backend receives high level trust data. Since your client application runs in a user's environment, you could not be sure about the integrity of the data if your client application were to receive high sensitive data from the AusweisApp2 directly as your backend does not have any possibility to verify the source of the data.

Also this approach, recommended for compliance reasons by the Federal Office for Information Security, spares your client application the necessity of encypting these high sensitive data.

In case your client application requires data input from the ID card, you need to get this from the backend system (e.g. the eID server) after a successful authentication.

See also:

TR-03124[1], part 1: Specifications

1.1 Recommended

The SDK uses JSON as the communication protocol. It is recommended to use an existing library for this.

- **Java**: Google GSON[2]
- **C++**: JsonCPP[3]

[1] https://www.bsi.bund.de/SharedDocs/Downloads/DE/BSI/Publikationen/TechnischeRichtlinien/TR03124/TR-03124-1.pdf

[2] https://github.com/google/gson

[3] https://github.com/open-source-parsers/jsoncpp

2 Android

This chapter deals with the Android specific properties of the AusweisApp2 SDK. The AusweisApp2 core is encapsulated into an **Android service** which is running in the background without a user interface. This service is interfaced via an Android specific interprocess communication (IPC) mechanism. The basics of this mechanism - the **Android Interface Definition Language** (AIDL) - are introduced in the following section. Subsequent sections deal with the SDK interface itself and explain which steps are necessary in order to talk to the AusweisApp2 SDK.

The AusweisApp2 is available as an integrated and as an external variant. The integrated version is provided as an AAR package that can automatically be fetched by Android's default build system **gradle**. The external variant is available as an APK in Google's PlayStore. It is required that the user has manually installed the AusweisApp2 like any other app to connect to the external variant.

Important: The integrated variant is available in jcenter for free. If you need enterprise support feel free to contact us.

2.1 Integrated

The integrated SDK is distributed as an AAR package that contains native **arm64-v8a** libraries only. The AAR package is available in the default repository of Android. The following listing shows the required **jcenter** in **build.gradle**.

```
buildscript {
    repositories {
        jcenter()
    }
}
```

The integrated AusweisApp2 will be fetched automatically as a dependency by your **app/build.gradle** file. It is recommended to always use the latest version by **1.+** of AusweisApp2. But you are free to add a concrete version like **1.16.0**.

```
dependencies {
    implementation 'com.governikus:ausweisapp:1.16.+'
}
```

See also:

The AAR package provides an **AndroidManifest.xml** to register required permissions and the background service. If your own AndroidManifest has conflicts with our provided file you can add some attributes to resolve those conflicts.

https://developer.android.com/studio/build/manifest-merge.html

2.2 External

The APK is available in Google's PlayStore and needs to be installed by the user. The external SDK is distributed as 32-bit and 64-bit.

Security

The following listing provides information about the solution to provide a secure connection to AusweisApp2.

- Data between two apps connected via AIDL as a bound service cannot be grabbed by an attacker. Android will send the data to the corresponding app directly. There is no broadcast like an implicit intent.
- An attacker cannot bind to an already bound service as AusweisApp2 will accept only one connection at the same time.
- An attacker cannot resume a connection after the previous app disconnects because AusweisApp2 will reset the internal state if an app connects with another session ID.
- An attacker cannot grab the session ID of the previous app because AusweisApp2 uses multiple sources of secure random number generator.
- An attacker cannot fake AusweisApp2 for other apps because the connection via AIDL is bound with package name "com.governikus.ausweisapp2". Google ensures that there is no other app in Google Play Store with that package name. Also the client app can check the fingerprint of signature certificate used for that package name.

Verify the authenticity of AusweisApp2

The following section deals with the cryptographic verification of the SDK's authenticity. This step is necessary to ensure that the SDK has not been modified in a malicious way.

Fingerprint

In order to verify that the AusweisApp2 SDK is authentic and has not been modified in a malicious way, it is required to verify its authenticity before establishing a connection with it. Each Android application is signed by a distribution certificate which ensures its authenticity. During the installation of an application Android verifies that it has been correctly signed with the supplied distribution certificate. So everything the client has to do in order to verify the authenticity is to verify that the correct certificate has been used. The **SHA256** fingerprint of the authentic SDK certificate is the following:

```
B0 2A C7 6B 50 A4 97 AE 81 0A EA C2 25 98 18 7B 3D 42 90 27 7D 08 51 A7 FA␣
↪8E 1A EA 5A 97 98 70
```

Example

The following example code demonstrates how the certificate hash value of a signed application on Android can be verified.

```
import android.content.pm.PackageInfo;
import android.content.pm.PackageManager;
import android.content.pm.Signature;

public class AusweisApp2Validator
{
```

(continues on next page)

(continued from previous page)

```
  private static final String PACKAGE = "com.governikus.ausweisapp2";
  private static final String FINGERPRINT = "..."; // see above

  public boolean isValid()
  {
    final PackageManager m = getPackageManager();
    PackageInfo info;
    try {
      info = m.getPackageInfo(PACKAGE, PackageManager.GET_SIGNATURES);
    } catch (PackageManager.NameNotFoundException e) {
      return false;
    }

    // What the API names signatures are actually the signing certificates.
    Signature certificates[] =  info.signatures;
    String computed = computeHashHexString("SHA256", certificates[0]);

    return FINGERPRINT.equalsIgnoreCase(computed);
  }
}
```

Import the AIDL files

Android provides an interprocess communication (IPC) mechanism which is based on messages consisting of primitive types. In order to abstract from this very basic mechanism, there is the Android Interface Definition Language (AIDL). It allows the definition of Java like interfaces. The Android SDK generates the necessary interface implementations from supplied AIDL files in order to perform IPC, as if this function had been called directly in the current process.

In order to interact with the AusweisApp2 SDK there are two AIDL interfaces. The first one is given to the client application by the SDK and allows the client to establish a session with the SDK, to send JSON commands to the SDK and to pass discovered NFC tags to the SDK.

The second AIDL interface is given to the SDK by the client application. It enables the client to receive the intial session parameters as well as JSON messages from the SDK. Furthermore it has a function which is called when an existing connection with the SDK is dropped by the SDK. Both interfaces are listed below and you need to import them into your build environment.

Important: It is required that you place the AIDL files under subdirectory “aidl/com.governikus.ausweisapp2”. Also the interface methods names must be exactly the same.

See also:

https://developer.android.com/guide/components/aidl.html

Note: If you implement the integrated variant beside the external variant you do **not** need to manually add AIDL files as the AAR package already provides those interfaces.

Interface

```
package com.governikus.ausweisapp2;

import com.governikus.ausweisapp2.IAusweisApp2SdkCallback;
import android.nfc.Tag;

interface IAusweisApp2Sdk
{
    boolean connectSdk(IAusweisApp2SdkCallback pCallback);
    boolean send(String pSessionId, String pMessageFromClient);
    boolean updateNfcTag(String pSessionId, in Tag pTag);
}
```

Callback

```
package com.governikus.ausweisapp2;

interface IAusweisApp2SdkCallback
{
    void sessionIdGenerated(String pSessionId, boolean pIsSecureSessionId);
    void receive(String pJson);
    void sdkDisconnected();
}
```

2.3 Background service

The integrated and external variants use the same method to establish a connection to the AusweisApp2 SDK. The AusweisApp2 SDK is a background service in the external AusweisApp2 or an embedded background service in your own application.

Binding to the service

In order to start the AusweisApp2 SDK it is necessary to bind to the Android service supplied by the SDK. This binding fulfils two purposes:

- First it starts the SDK.
- Second it enables the client to establish an IPC connection as mentioned above.

Due to the nature of an Android service, there can be only one instance of the SDK running. If multiple clients bind to the service, they are interacting with the same instance of the service. The service is terminated once all previously bound clients are unbound.

To differentiate between different connected clients, virtual sessions are used once the binding is completed. These sessions are discussed in a separate section, section *Create session to AusweisApp2* (page 8).

Create connection

First of all, in order to bind to the service, one needs to instantiate an Android ServiceConnection. Subsequently, the object is passed to the Android API and the contained methods are invoked by Android on service connection and disconnection.

```
import android.content.ServiceConnection;

// [...]

ServiceConnection mConnection = new ServiceConnection()
{
  @Override
  public void onServiceConnected(ComponentName className, IBinder service)
  {
      // ... details below
  }

  @Override
  public void onServiceDisconnected(ComponentName className)
  {
      // ... details below
  }
}
```

Bind service to raw connection

In order to perform the actual binding a directed Intent, which identifies the AusweisApp2 SDK, is created. This Intent is sent to the Android API along with the ServiceConnection created above. This API call either starts up the SDK if it is the first client, or connects to the running SDK instance if there is already another client bound.

If you use the external variant of AusweisApp2 you need to pass the package name of Governikus. Otherwise you need to pass your own package name as the integrated variant is a background service of your application.

```
import android.app.Activity;
import android.content.Context;
import android.content.Intent;

// [...]

String pkg = "com.governikus.ausweisapp2";

boolean useIntegrated = true; // use external or integrated
if (useIntegrated)
  pkg = getApplicationContext().getPackageName();

String name = "com.governikus.ausweisapp2.START_SERVICE";
Intent serviceIntent = new Intent(name);
serviceIntent.setPackage(pkg);
bindService(serviceIntent, mConnection, Context.BIND_AUTO_CREATE);
```

See also:

https://developer.android.com/guide/components/bound-services.html

https://developer.android.com/reference/android/app/Activity.html

Redirect to Play Store

It is necessary that AusweisApp2 is installed in order to use the external SDK. It is recommended to check and display a message in case the user needs to install AusweisApp2 first. Also, the user should be redirected to the Play Store entry to find the app.

```
import android.content.ActivityNotFoundException;
import android.content.pm.ResolveInfo;
import android.net.Uri;
import java.util.List;

PackageManager m = getPackageManager();
List<ResolveInfo> list = m.queryIntentServices(serviceIntent,␣
↪PackageManager.MATCH_ALL);

if (list == null || list.isEmpty())
{
  final String name = "com.governikus.ausweisapp2";
  try {
    startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("market://
↪details?id=" + name)));
  } catch (ActivityNotFoundException e) {
    // Use the browser if Play Store is not installed, too!
    startActivity(new Intent(Intent.ACTION_VIEW, Uri.parse("https://play.
↪google.com/store/apps/details?id=" + name)));
  }
}
```

Note: This is not necessary if you use the integrated variant.

Initializing the AIDL connection

Once the Android service of the AusweisApp2 SDK is successfully started and bound to by the client, the Android system calls the onServiceConnected method of the ServiceConnection created and supplied above. This method receives an instance of the IBinder Android service interface.

The IBinder is then used by the client application to initialize the auto generated AIDL stub in order to use the AIDL IPC mechanism. The used stub is supposed to be auto generated by the Android SDK if you have properly configured your build environment.

The stub initialization returns an instance of **IAusweisApp2Sdk** which is used to interact with the SDK. The example below stores this instance in the member variable mSdk.

```
import android.content.ComponentName;
import android.content.ServiceConnection;
import android.os.IBinder;

import com.governikus.ausweisapp2.IAusweisApp2Sdk;
```

(continues on next page)

(continued from previous page)

```
// [...]

IAusweisApp2Sdk mSdk;

ServiceConnection mConnection = new ServiceConnection(){
  @Override
  public void onServiceConnected(ComponentName className, IBinder service)
  {
      try {
          mSdk = IAusweisApp2Sdk.Stub.asInterface(service);
      } catch (ClassCastException|RemoteException e) {
          // ...
      }
  }

  @Override
  public void onServiceDisconnected(ComponentName className)
  {
      mSdk = null;
  }
}
```

See also:

Import the AIDL files (page 4)

Create session to AusweisApp2

Once your client is bound to the AusweisApp2 SDK service and you have initialized the AIDL IPC mechanism, you are ready to use the actual SDK API.

Since the Android system does not allow to limit the number of clients which can connect to a service, the SDK API uses custom **sessions** to manage the connected clients. There is a maximum of one established session at a time.

In order to open a session with the SDK you need to pass an instance of **IAusweisApp2SdkCallback** to the **connectSdk** function of your previously acquired instance of **IAusweisApp2Sdk**. If your callback is accepted, the function returns true. Otherwise there is a problem with your supplied callback. Sessions will be disconnected once the IBinder instance of the connected client is invalidated, another communication error occurs or another Client connects. Please see *Disconnect from SDK* (page 10) for instructions to gracefully disconnect from the SDK.

As mentioned above: If there already is a connected client and a second client attempts to connect, the first client is disconnected and the second client is granted exclusive access to the SDK. The first client is informed via its callback by **sdkDisconnected**. The second client is presented a fresh environment and it has no access to any data of the first client.

If you have successfully established a session, the **sessionIdGenerated** function of your callback is invoked. With this invocation you receive two arguments. **pIsSecureSessionId** is true if the SDK was able to gather enough entropy in order to generate a secure random session ID. If it is false, there is no session ID passed. There is nothing you can do about such an error. It results from a problem with the random number generator, which in turn is very likely the result of an ongoing local attack. The device should be considered manipulated and the user should be informed.

On success **pSessionId** holds the actual session ID generated by the SDK. This ID is used to identify your session and you need to pass it to all future SDK function invocations of this session.

The listing below shows an example for an instantiation of IAusweisApp2SdkCallback and establishing a session.

```
import com.governikus.ausweisapp2.IAusweisApp2Sdk;
import com.governikus.ausweisapp2.IAusweisApp2SdkCallback;

// [...]

LocalCallback mCallback = new LocalCallback();
class LocalCallback extends IAusweisApp2SdkCallback.Stub
{
  public String mSessionID = null;

  @Override
  public void sessionIdGenerated(
    String pSessionId, boolean pIsSecureSessionId) throws RemoteException
  {
      mSessionID = pSessionId;
  }

  @Override
  public void receive(String pJson) throws RemoteException
  {
      // handle message from SDK
  }
}

// [...]

try
{
  if (!mSdk.connectSdk(mCallback))
  {
      // already connected? Handle error...
  }
}
catch (RemoteException e)
{
    // handle exception
}
```

See also:

Initializing the AIDL connection (page 7) *Disconnect from SDK* (page 10)

Send command

In order to send a JSON command to the AusweisApp2 SDK, you need to invoke the **send** function of your instance of **IAusweisApp2Sdk**. For this command to be processed by the SDK you need to supply the session ID which you have previously received. The listing below shows an example.

```
String cmd = "{\"cmd\": \"GET_INFO\"}";
try
```

(continues on next page)

(continued from previous page)

```
{
  if (!mSdk.send(mCallback.mSessionID, cmd))
  {
      // disconnected? Handle error...
  }
}
catch (RemoteException e)
{
    // handle exception
}
```

Receive message

Messages from the AusweisApp2 SDK are passed to you via the same instance of **IAusweisApp2SdkCallback** in which you have received the session ID. The **receive** method is called each time the SDK sends a message.

See also:

Create session to AusweisApp2 (page 8)

Disconnect from SDK

In order to disconnect from the AusweisApp2 SDK you need to invalidate your instance of **IBinder**. There are two possibilities to do this. The first one is to unbind from the SDK Android service to undo your binding, like shown in the code listing below. The second one is to return false in the **pingBinder** function of your IBinder instance.

```
unbindService(mConnection);
```

See also:

Binding to the service (page 5)

https://developer.android.com/reference/android/os/IBinder.html

Passing NFC tags to the SDK

NFC tags can only be detected by applications which have a foreground window on the Android platform. A common workaround for this problem is to equip background services with a transparent window which is shown to dispatch NFC tags.

However, if there are multiple applications installed, which are capable of dispatching NFC tags, the Android system will display an **App Chooser** for each discovered tag enabling the user to select the appropriate application to handle the NFC tag. To have such a chooser display the name and image of the client application instead of the SDK, the client application is required to dispatch discovered NFC tags and to pass them to the SDK.

Furthermore, this interface design enables the client application to do **foreground dispatching** of NFC tags. If the active application registers itself for foreground dispatching, it receives discovered NFC tags directly without Android displaying an App Chooser.

Permissions in AndroidManifest.xml

The client applications needs to register the NFC permission as shown in the listing below in order to access the NFC reader hardware.

```
<uses-permission android:name="android.permission.NFC"/>
```

See also:

https://developer.android.com/guide/topics/security/permissions.html

---

Note: The integrated variant already provides an **AndroidManifest.xml** with prepared permissions.

Intent-Filter in AndroidManifest.xml

In order to be informed about attached NFC tags by Android, the client application is required to register an intent filter. The appropriate filter is shown in the listing below.

```
<intent-filter>
  <action android:name="android.nfc.action.TECH_DISCOVERED" />
</intent-filter>
<meta-data android:name="android.nfc.action.TECH_DISCOVERED"␣
↪android:resource="@xml/nfc_tech_filter" />
```

See also:

https://developer.android.com/guide/components/intents-filters.html

NFC Technology Filter

Since there are many different kinds of NFC tags, Android requires the application to register a technology filter for the kind of tags the application wants to receive. The proper filter for the German eID card is shown in the listing below.

```
<resources xmlns:xliff="urn:oasis:names:tc:xliff:document:1.2">
  <tech-list>
    <tech>android.nfc.tech.IsoDep</tech>
  </tech-list>
</resources>
```

Implementation

As it is common on the Android platform, information is send to applications encapsulated in instances of the **Intent** class. In order to process newly discovered NFC tags, Intents which are given to the application need to be checked for the parcelable NFC extra as shown in the code listing below. Subsequently the client is required to send them to the AusweisApp2 SDK by calling the **updateNfcTag** method (of) the previously acquired **IAusweisApp2Sdk** instance. The listing below shows an example for the described process.

```
import android.content.Intent;
import android.nfc.NfcAdapter;
import android.nfc.Tag;

import com.governikus.ausweisapp2.IAusweisApp2Sdk;
import com.governikus.ausweisapp2.IAusweisApp2SdkCallback;

// [...]

void handleIntent(Intent intent)
{
  final Tag tag = intent.getParcelableExtra(NfcAdapter.EXTRA_TAG);
  if (tag != null)
  {
    try {
      mSdk.updateNfcTag(mCallback.mSessionID, tag);
    } catch (RemoteException e) {
      // ...
    }
  }
}
```

Dispatching NFC tags in foreground

As already mentioned under *Passing NFC tags to the SDK* (page 10), an App Chooser is displayed for discovered NFC tags by Android if multiple applications which are able to dispatch NFC tags are installed. An application can suppress this App Chooser if it registers itself for **foreground dispatching** at runtime. This way NFC tags are send directly to the registered application without a chooser being displayed. An example implementation of the required steps in order to register are shown in code listing below.

```
import android.app.Activity;
import android.nfc.NfcAdapter;
import android.content.Intent;
import android.app.PendingIntent;
import android.content.IntentFilter;
import android.nfc.tech.IsoDep;

class ForegroundDispatcher
{
  private final Activity mActivity;
  private final NfcAdapter mAdapter;
  private final PendingIntent mPendingIntent;
  private final IntentFilter[] mFilters;
  private final String[][] mTechLists;

  ForegroundDispatcher(Activity pActivity)
  {
    mActivity = pActivity;
    mAdapter = NfcAdapter.getDefaultAdapter(mActivity);
    Intent intent = new Intent(mActivity, mActivity.getClass()).
      addFlags(Intent.FLAG_ACTIVITY_SINGLE_TOP);
    mPendingIntent = PendingIntent.getActivity(mActivity, 0, intent, 0);
```

(continues on next page)

(continued from previous page)

```
    mFilters = new IntentFilter[] {
            new IntentFilter(NfcAdapter.ACTION_TECH_DISCOVERED)
    };
    mTechLists = new String[][] { new String[] {
            IsoDep.class.getName()
    } };
  }

  void enable()
  {
    if (mAdapter != null)
      mAdapter.enableForegroundDispatch(mActivity,
                                        mPendingIntent,
                                        mFilters,
                                        mTechLists);
  }

  void disable()
  {
    if (mAdapter != null)
      mAdapter.disableForegroundDispatch(mActivity);
  }
}

// [...]

ForegroundDispatcher mDispatcher = new ForegroundDispatcher(this);
```

The example implementation from above needs to be invoked when the application is brought to foreground and when it looses focus. An example for appropriate places are the **onResume** and the **onPause** methods of Activities as shown in the code listing below.

```
@Override
public void onResume()
{
  super.onResume();
  mDispatcher.enable();
}

@Override
public void onPause()
{
  super.onPause();
  mDispatcher.disable();
}
```

See also:

https://developer.android.com/reference/android/app/Activity.html#ActivityLifecycle

3 Desktop

This chapter deals with the desktop specific properties of the AusweisApp2 SDK. The AusweisApp2 core is reachable over a **WebSocket** which is running by default since AusweisApp2 1.16.0. Subsequent

sections deal with the SDK interface itself and explain which steps are necessary in order to talk to the AusweisApp2 SDK.

3.1 WebSocket

The AusweisApp2 uses the same default port as defined in TR-03124-1. Your application can connect to `ws://localhost:24727/eID-Kernel` to establish a bidirectional connection.

You can check the version of AusweisApp2 by the `Server` header of the HTTP response or by an additional query to get the *Status* (page 14).

If the WebSocket handshake was succesful your application can send *Commands* (page 16) and receive *Messages* (page 21). The AusweisApp2 will send an HTTP error 503 "Service Unavailable" if the WebSocket is disabled.

See also:

https://tools.ietf.org/html/rfc6455

User installed

Your application can connect to a user installed AusweisApp2. If the user already has an active workflow your request will be refused by an HTTP error 409 "Conflict". Also it is not possible to connect multiple times to the WebSocket as only one connection is allowed and will be refused by an HTTP error 429 "Too Many Requests". Once an application is connected to the WebSocket the graphical user interface of the AusweisApp2 will be blocked and shows a hint that another application uses the AusweisApp2.

Important: Please provide a `User-Agent` in your HTTP upgrade request! The AusweisApp2 will show the content to the user as a hint which application uses the AusweisApp2.

Integrated

You can deliver separate AusweisApp2 binaries inside your own application or start an already installed AusweisApp2. If your application spawns a separate process you should provide the cmdline parameter `--port 0` to avoid conflicts with a user started AusweisApp2 and other processes that uses a specified port.

The AusweisApp2 will create a text file in the system temporary directory to provide the selected port. The port filename contains the PID of the running process to allow multiple instances at the same time.

Example: **$TMPDIR/AusweisApp2.12345.port**

Your application can avoid the graphical interface of AusweisApp2 by providing the cmdline parameter `--ui websocket`.

3.2 Status

TR-03124-1 defined a query for status information. This is useful to fetch current version of installed AusweisApp2 to check if the version supports the WebSocket-API.

You can get this by a HTTP GET query to `http://localhost:24727/eID-Client?Status`. If you prefer the JSON syntax you can add it to the parameter `?Status=json` to get the following information.

```
{
  "Implementation-Title": "AusweisApp2",
  "Implementation-Vendor": "Governikus GmbH & Co. KG",
  "Implementation-Version": "1.16.0",
  "Name": "AusweisApp2",
  "Specification-Title": "TR-03124",
  "Specification-Vendor": "Federal Office for Information Security",
  "Specification-Version": "1.3"
}
```

See also:

The AusweisApp2 SDK provides a *GET_INFO* (page 16) command and an *INFO* (page 28) message to fetch the same information to check the compatibility of used AusweisApp2.

3.3 Reader

The AusweisApp2 SDK uses PC/SC and paired Smartphones as card reader. If the user wants to use the “smartphone as card reader” feature it is necessary to pair the devices by the graphical interface of AusweisApp2. The AusweisApp2 SDK provides no API to pair those devices.

4 Commands

Your application (client) can send some commands (**cmd**) to control the AusweisApp2. The AusweisApp2 (server) will send some proper *Messages* (page 21) during the whole workflow or as an answer to your command.

4.1 GET_INFO

Returns information about the current installation of AusweisApp2.

The AusweisApp2 will send an *INFO* (page 28) message as an answer.

```
{"cmd": "GET_INFO"}
```

4.2 GET_API_LEVEL

Returns information about the available and current API level.

The AusweisApp2 will send an *API_LEVEL* (page 23) message as an answer.

```
{"cmd": "GET_API_LEVEL"}
```

4.3 SET_API_LEVEL

Set supported API level of your application.

If you initially develop your application against the AusweisApp2 SDK you should check with *GET_API_LEVEL* (page 16) the highest supported level and set this value with this command if you connect to the SDK. This will set the SDK to act with the defined level even if a newer level is available.

The AusweisApp2 will send an *API_LEVEL* (page 23) message as an answer.

- **level**: Supported API level of your app.

```
{
  "cmd": "SET_API_LEVEL",
  "level": 1
}
```

4.4 GET_READER

Returns information about the requested reader.

If you explicitly want to ask for information of a known reader name you can request it with this command.

The AusweisApp2 will send a *READER* (page 30) message as an answer.

- **name**: Name of the reader.

```
{
  "cmd": "GET_READER",
  "name": "NAME OF THE READER"
}
```

4.5 GET_READER_LIST

Returns information about all connected readers.

If you explicitly want to ask for information of all connected readers you can request it with this command.

The AusweisApp2 will send a *READER_LIST* (page 30) message as an answer.

```
{"cmd": "GET_READER_LIST"}
```

4.6 RUN_AUTH

Starts an authentication.

The AusweisApp2 will send an *AUTH* (page 23) message when the authentication is started.

- **tcTokenURL**: URL to the TcToken. This is equal to the desktop style activation URL. *(http://127.0.0.1:24727/eID-Client?tcTokenURL=)*

```
{
  "cmd": "RUN_AUTH",
  "tcTokenURL": "https://test.governikus-eid.de/Autent-DemoApplication/
↪RequestServlet?provider=demo_epa_20&redirect=true"
}
```

Note: This command is allowed only if the AusweisApp2 has no running authentication. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.7 GET_ACCESS_RIGHTS

Returns information about the requested access rights.

The AusweisApp2 will send an *ACCESS_RIGHTS* (page 21) message as an answer.

```
{"cmd": "GET_ACCESS_RIGHTS"}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ACCESS_RIGHTS* (page 21) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.8 SET_ACCESS_RIGHTS

Set effective access rights.

By default the **effective** access rights are **optional** + **required**. If you want to enable or disable some **optional** access rights you can send this command to modify the **effective** access rights.

The AusweisApp2 will send an *ACCESS_RIGHTS* (page 21) message as an answer.

- **chat**: List of enabled **optional** access rights. If you send an empty **[]** all **optional** access rights are disabled.

```
{
  "cmd": "SET_ACCESS_RIGHTS",
  "chat": []
}
```

```
{
  "cmd": "SET_ACCESS_RIGHTS",
  "chat": ["FamilyName"]
}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ACCESS_RIGHTS* (page 21) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

See also:

List of possible access rights are listed in *ACCESS_RIGHTS* (page 21).

4.9 GET_CERTIFICATE

Returns the certificate of current authentication.

The AusweisApp2 will send a *CERTIFICATE* (page 25) message as an answer.

```
{"cmd": "GET_CERTIFICATE"}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ACCESS_RIGHTS* (page 21) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.10 CANCEL

Cancel the whole workflow.

If your application sends this command the AusweisApp2 will cancel the workflow. You can send this command in any state of a running workflow to abort it.

```
{"cmd": "CANCEL"}
```

Note: This command is allowed only if the AusweisApp2 started an authentication. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.11 ACCEPT

Accept the current state.

If the AusweisApp2 will send the message *ACCESS_RIGHTS* (page 21) the user needs to **accept** or **deny**. So the workflow is paused until your application sends this command to accept the requested information.

If the user does not accept the requested information your application needs to send the command *CANCEL* (page 18) to abort the whole workflow.

This command will be used later for additional requested information if the AusweisApp2 needs to pause the workflow. In *API_LEVEL* (page 23) v1 only *ACCESS_RIGHTS* (page 21) needs to be accepted.

```
{"cmd": "ACCEPT"}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ACCESS_RIGHTS* (page 21) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.12 SET_PIN

Set PIN of inserted card.

If the AusweisApp2 sends message *ENTER_PIN* (page 26) you need to send this command to unblock the card with the PIN.

The AusweisApp2 will send an *ENTER_PIN* (page 26) message on error or message *ENTER_CAN* (page 25) if the retryCounter of the card is decreased to **1**. For detailed information see message *ENTER_PIN* (page 26).

If the PIN was correct, the workflow will continue.

If the last attempt to enter the PIN failed, AusweisApp2 will send the message *ENTER_PUK* (page 27) as the retryCounter is decreased to **0**.

Changed in version 1.16.0: The parameter “value” must be omitted if the used *READER* (page 30) has a keypad.

- **value**: The personal identification number (PIN) of the card. This must be 6 digits if the *READER* (page 30) has no keypad, otherwise this parameter must be omitted.

```
{
  "cmd": "SET_PIN",
  "value": "123456"
}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ENTER_PIN* (page 26) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.13 SET_CAN

Set CAN of inserted card.

If the AusweisApp2 sends message *ENTER_CAN* (page 25) you need to send this command to unblock the last retry of *SET_PIN* (page 19).

The AusweisApp2 will send an *ENTER_CAN* (page 25) message on error. Otherwise the workflow will continue with *ENTER_PIN* (page 26).

Changed in version 1.16.0: The parameter "value" must be omitted if the used *READER* (page 30) has a keypad.

- **value**: The card access number (CAN) of the card. This must be 6 digits if the *READER* (page 30) has no keypad, otherwise this parameter must be omitted.

```
{
  "cmd": "SET_CAN",
  "value": "123456"
}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ENTER_CAN* (page 25) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

4.14 SET_PUK

Set PUK of inserted card.

If the AusweisApp2 sends message *ENTER_PUK* (page 27) you need to send this command to unblock *SET_PIN* (page 19).

The AusweisApp2 will send an *ENTER_PUK* (page 27) message on error or if the PUK is operative. Otherwise the workflow will continue with *ENTER_PIN* (page 26). For detailed information see message *ENTER_PUK* (page 27).

Changed in version 1.16.0: The parameter "value" must be omitted if the used *READER* (page 30) has a keypad.

- **value**: The personal unblocking key (PUK) of the card. This must be 10 digits if the *READER* (page 30) has no keypad, otherwise this parameter must be omitted.

```
{
  "cmd": "SET_PUK",
  "value": "1234567890"
}
```

Note: This command is allowed only if the AusweisApp2 sends an initial *ENTER_PUK* (page 27) message. Otherwise you will get a *BAD_STATE* (page 24) message as an answer.

5 Messages

The AusweisApp2 (server) will send some proper messages (**msg**) to your application (client) during the whole workflow or as an answer to your *Commands* (page 16).

5.1 ACCESS_RIGHTS

This message will be send by AusweisApp2 once the authentication is started by *RUN_AUTH* (page 17) and the AusweisApp2 got the certificate from the service.

If your application receives this message you can call *SET_ACCESS_RIGHTS* (page 18) to change some optional access rights or you can call *GET_ACCESS_RIGHTS* (page 17) to get this message again.

Also you can call *GET_CERTIFICATE* (page 18) to get the certificate to show this to your user.

The workflow will continue if you call *ACCEPT* (page 19) to indicate that the user accepted the requested access rights or call *CANCEL* (page 18) to abort the whole workflow.

- **error**: This optional parameter indicates an error of a *SET_ACCESS_RIGHTS* (page 18) call if the command contained invalid data.
- **aux**: Optional auxiliary data of the service provider.
  - **ageVerificationDate**: Optional required date of birth for AgeVerification as ISO 8601.
  - **requiredAge**: Optional required age for AgeVerification. It is calculated by AusweisApp2 on the basis of ageVerificationDate and current date.
  - **validityDate**: Optional validity date as ISO 8601.
  - **communityId**: Optional id of community.
- **chat**: Access rights of the service provider.
  - **effective**: Indicates the enabled access rights of **optional** and **required**.
  - **optional**: These rights are optional and can be enabled or disabled by *SET_ACCESS_RIGHTS* (page 18).
  - **required**: These rights are mandatory and cannot be disabled.
- **transactionInfo**: Optional transaction information.

```
{
  "msg": "ACCESS_RIGHTS",
  "error": "some optional error message",
  "aux":
        {
         "ageVerificationDate": "1999-07-20",
         "requiredAge": "18",
         "validityDate": "2017-07-20",
         "communityId": "02760400110000"
```

(continues on next page)

(continued from previous page)

```
        },
  "chat":
         {
          "effective": ["Address", "FamilyName", "GivenNames",
↪"AgeVerification"],
          "optional": ["GivenNames", "AgeVerification"],
          "required": ["Address", "FamilyName"]
         },
  "transactionInfo": "this is an example"
}
```

```
{
  "msg": "ACCESS_RIGHTS",
  "chat":
         {
          "effective": ["Address", "FamilyName", "GivenNames",
↪"AgeVerification"],
          "optional": ["GivenNames", "AgeVerification"],
          "required": ["Address", "FamilyName"]
         }
}
```

The following access rights are possible:

- Address
- BirthName
- FamilyName
- GivenNames
- PlaceOfBirth
- DateOfBirth
- DoctoralDegree
- ArtisticName
- Pseudonym
- ValidUntil
- Nationality
- IssuingCountry
- DocumentType
- ResidencePermitI
- ResidencePermitII
- CommunityID
- AddressVerification
- AgeVerification

See also:

TR-03110[4], part 4, chapter 2.2.3

TR-03127[5], chapter 3.2.2

5.2 API_LEVEL

This message will be send if *GET_API_LEVEL* (page 16) or *SET_API_LEVEL* (page 16) is called.

It lists all **available** API levels that can be used and set by *SET_API_LEVEL* (page 16). Also it indicates the **current** selected API level.

- **error**: Optional error message if *SET_API_LEVEL* (page 16) failed.
- **available**: List of supported API level by this version.
- **current**: Currently selected API level.

```
{
  "msg": "API_LEVEL",
  "error": "optional error message like an invalid level",
  "available": [1,2,3,4],
  "current": 4
}
```

Your application should always set the compatible API level. The AusweisApp2 will support multiple API levels to give you enough time to add support for the new API.

Even if you added support for the new API, your application should still support the old API level in case the user updates your application but does not update the AusweisApp2. Otherwise you need to show a message to the user that they need to update the AusweisApp2.

The API level will be increased for **incompatible** changes only. If we can add additional commands, messages or information without breaking the previous API you can check the application version with *GET_INFO* (page 16) to know if the current version supports your requirements.

This documentation will mark every API change with a flag like the following:

- New in version 1.10.0.
- Changed in version 1.10.0.
- Deprecated since version 1.10.0.

5.3 AUTH

This message will be send by AusweisApp2 if an authentication is initially started. The next message should be *ACCESS_RIGHTS* (page 21) or *AUTH* (page 23) again if the authentication immediately results in an error.

If you receive an *AUTH* (page 23) message with a parameter **error** your command *RUN_AUTH* (page 17) was invalid and the workflow was not started at all.

- **error**: Optional error message if *RUN_AUTH* (page 17) failed.

[4] https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html
[5] https://www.bsi.bund.de/DE/Publikationen/TechnischeRichtlinien/tr03127/tr-03127.html

```
{
  "msg": "AUTH",
  "error": "error message if RUN_AUTH failed"
}
```

If the workflow is finished the AusweisApp2 will send a message with a result and an url parameter to indicate the end of an authentication.

- **result**: The final result of authentication.
  - **major**: Major "error" code.
  - **minor**: Minor error code.
  - **language**: Language of description and message. AusweisApp2 will use "de" for German system locale or "en" as the fall back.
  - **description**: Description of the error message.
  - **message**: The error message.
- **url**: Refresh url or communication error address.

```
{
  "msg": "AUTH",
  "result":
            {
             "major": "http://www.bsi.bund.de/ecard/api/1.1/resultmajor#ok"
            },
  "url": "https://test.governikus-eid.de/gov_autent/async?refID=_123456789"
}
```

```
{
  "msg": "AUTH",
  "result":
            {
             "major": "http://www.bsi.bund.de/ecard/api/1.1/resultmajor
↪#error",
             "minor": "http://www.bsi.bund.de/ecard/api/1.1/resultminor/al/
↪common#internalError",
             "language": "en",
             "description": "An internal error has occurred during␣
↪processing.",
             "message": "The ID card has been removed. The process is␣
↪aborted."
            },
  "url": "https://test.governikus-eid.de/gov_autent/async?refID=_abcdefgh"
}
```

5.4 BAD_STATE

Indicates that your previous command was send in an invalid state.

Some commands can be send to the server only if certain "state" is reached in the workflow to obtain the corresponding result. Otherwise the command will fail with *BAD_STATE* (page 24).

For example, you cannot send *GET_CERTIFICATE* (page 18) if there is no authentication in progress.

- **error**: Name of the received command that is invalid in this state.

```
{
  "msg": "BAD_STATE",
  "error": "NAME_OF_YOUR_COMMAND"
}
```

5.5 CERTIFICATE

Provides information about the used certificate.

- **description**: Detailed description of the certificate.
  - **issuerName**: Name of the certificate issuer.
  - **issuerUrl**: URL of the certificate issuer.
  - **subjectName**: Name of the certificate subject.
  - **subjectUrl**: URL of the certificate subject.
  - **termsOfUsage**: Raw certificate information about the terms of usage.
  - **purpose**: Parsed purpose of the terms of usage.
- **validity**: Validity dates of the certificate in UTC.
  - **effectiveDate**: Certificate is valid since this date.
  - **expirationDate**: Certificate is invalid after this date.

```
{
  "msg": "CERTIFICATE",
  "description":
                {
                 "issuerName": "Governikus Test DVCA",
                 "issuerUrl": "http://www.governikus.de",
                 "subjectName": "Governikus GmbH & Co. KG",
                 "subjectUrl": "https://test.governikus-eid.de",
                 "termsOfUsage": "Anschrift:\t\r\nGovernikus GmbH & Co.␣
↪KG\r\nAm Fallturm 9\r\n28359 Bremen\t\r\n\r\nE-Mail-Adresse:\thb@bos-
↪bremen.de\t\r\n\r\nZweck des Auslesevorgangs:\tDemonstration des eID-
↪Service\t\r\n\r\nZuständige Datenschutzaufsicht:\t\r\nDie␣
↪Landesbeauftragte für Datenschutz und Informationsfreiheit der Freien␣
↪Hansestadt Bremen\r\nArndtstraße 1\r\n27570 Bremerhaven",
                 "purpose": "Demonstration des eID-Service"
                },
  "validity":
              {
               "effectiveDate": "2016-07-31",
               "expirationDate": "2016-08-30"
              }
}
```

5.6 ENTER_CAN

Indicates that a CAN is required to continue workflow.

If the AusweisApp2 sends this message, you will have to provide the CAN of the inserted card with *SET_CAN* (page 20).

The CAN is required to enable the last attempt of PIN input if the retryCounter is **1**. The workflow continues automatically with the correct CAN and the AusweisApp2 will send an *ENTER_PIN* (page 26) message. Despite the correct CAN being entered, the retryCounter remains at **1**.

The CAN is also required, if the authentication terminal has an approved "CAN allowed right". This allows the workflow to continue without an additional PIN.

If your application provides an invalid *SET_CAN* (page 20) command the AusweisApp2 will send an *ENTER_CAN* (page 25) message with an error parameter.

If your application provides a valid *SET_CAN* (page 20) command and the CAN was incorrect the AusweisApp2 will send *ENTER_CAN* (page 25) again but without an error parameter.

New in version 1.14.2: Support of "CAN allowed right".

- **error**: Optional error message if your command *SET_CAN* (page 20) was invalid.
- **reader**: Information about the used card and card reader. Please see message *READER* (page 30) for details.

```
{
  "msg": "ENTER_CAN",
  "error": "You must provide 6 digits",
  "reader":
            {
             "name": "NFC",
             "attached": true,
             "keypad": false,
             "card":
                     {
                      "inoperative": false,
                      "deactivated": false,
                      "retryCounter": 1
                     }
            }
}
```

Note: There is no retry limit for an incorrect CAN.

5.7 ENTER_PIN

Indicates that a PIN is required to continue the workflow.

If the AusweisApp2 sends this message, you will have to provide the PIN of the inserted card with *SET_PIN* (page 19).

The workflow will automatically continue if the PIN was correct. Otherwise you will receive another message *ENTER_PIN* (page 26). If the correct PIN is entered the retryCounter will be set to **3**.

If your application provides an invalid *SET_PIN* (page 19) command the AusweisApp2 will send an *ENTER_PIN* (page 26) message with an error parameter and the retryCounter of the card is **not** decreased.

If your application provides a valid *SET_PIN* (page 19) command and the PIN was incorrect the AusweisApp2 will send *ENTER_PIN* (page 26) again with a decreased retryCounter but without an error parameter.

If the value of retryCounter is **1** the AusweisApp2 will initially send an *ENTER_CAN* (page 25) message. Once your application provides a correct CAN the AusweisApp2 will send an *ENTER_PIN* (page 26) again with a retryCounter of **1**.

If the value of retryCounter is **0** the AusweisApp2 will initially send an *ENTER_PUK* (page 27) message. Once your application provides a correct PUK the AusweisApp2 will send an *ENTER_PIN* (page 26) again with a retryCounter of **3**.

- **error**: Optional error message if your command *SET_PIN* (page 19) was invalid.
- **reader**: Information about the used card and card reader. Please see message *READER* (page 30) for details.


```
{
  "msg": "ENTER_PIN",
  "error": "You must provide 6 digits",
  "reader":
            {
             "name": "NFC",
             "attached": true,
             "keypad": false,
             "card":
                     {
                      "inoperative": false,
                      "deactivated": false,
                      "retryCounter": 3
                     }
            }
}
```


5.8 ENTER_PUK

Indicates that a PUK is required to continue the workflow.

If the AusweisApp2 sends this message, you will have to provide the PUK of the inserted card with *SET_PUK* (page 20).

The workflow will automatically continue if the PUK was correct and the AusweisApp2 will send an *ENTER_PIN* (page 26) message. Otherwise you will receive another message *ENTER_PUK* (page 27). If the correct PUK is entered the retryCounter will be set to **3**.

If your application provides an invalid *SET_PUK* (page 20) command the AusweisApp2 will send an *ENTER_PUK* (page 27) message with an error parameter.

If your application provides a valid *SET_PUK* (page 20) command and the PUK was incorrect the AusweisApp2 will send *ENTER_PUK* (page 27) again but without an error parameter.

If AusweisApp2 sends *ENTER_PUK* (page 27) with field “inoperative” of embedded *READER* (page 30) message set true it is not possible to unblock the PIN. You will have to show a message to the user that the card is inoperative and the user should contact the authority responsible for issueing the identification document to unblock the PIN. You need to send a *CANCEL* (page 18) to abort the workflow if card is operative. Please see the note for more information.

- **error**: Optional error message if your command *SET_PUK* (page 20) was invalid.
- **reader**: Information about the used card and card reader. Please see message *READER* (page 30) for details.


```
{
  "msg": "ENTER_PUK",
  "error": "You must provide 10 digits",
  "reader":
            {
             "name": "NFC",
             "attached": true,
             "keypad": false,
             "card":
                    {
                     "inoperative": false,
                     "deactivated": false,
                     "retryCounter": 0
                    }
            }
}
```

Note: There is no retry limit for an incorrect PUK. But be aware that the PUK can only be used 10 times to unblock the PIN. There is no readable counter for this. The AusweisApp2 is not able to provide any counter information of PUK usage. If the PUK is used 10 times it is not possible to unblock the PIN anymore and the card will remain in PUK state. Also it is not possible to indicate this state before the user enters the correct PUK once. This information will be provided as field “inoperative” of *READER* (page 30) message.

5.9 INFO

Provides information about the AusweisApp2.

Especially if you want to get a specific **Implementation-Version** to check if the current installation supports some additional *Commands* (page 16) or *Messages* (page 21).

Also you should check the *API_LEVEL* (page 23) as it will be increased for **incompatible** changes.

- **VersionInfo**: Structure of version information.
  - **Name**: Application name.
  - **Implementation-Title**: Title of implementation.
  - **Implementation-Vendor**: Vendor of implementation.
  - **Implementation-Version**: Version of implementation.
  - **Specification-Title**: Title of specification.
  - **Specification-Vendor**: Vendor of specification.
  - **Specification-Version**: Version of specification.


```
{
  "msg": "INFO",
```


(continues on next page)

(continued from previous page)

```
  "VersionInfo":
                {
                 "Name": "AusweisApp2",
                 "Implementation-Title": "AusweisApp2",
                 "Implementation-Vendor": "Governikus GmbH & Co. KG",
                 "Implementation-Version": "1.10.0",
                 "Specification-Title": "TR-03124",
                 "Specification-Vendor": "Federal Office for Information␣
↪Security",
                 "Specification-Version": "1.2"
                }
}
```

5.10 INSERT_CARD

Indicates that the AusweisApp2 requires a card to continue.

If the AusweisApp2 needs a card to continue the workflow this message will be send as a notification. If your application receives this message it should show a hint to the user.

After the user inserted a card the workflow will automatically continue, unless the eID functionality is disabled. In this case, the workflow will be paused until another card is inserted. If the user already inserted a card this message will not be sent at all.

This message will also be send if there is no connected card reader.

```
{"msg": "INSERT_CARD"}
```

5.11 INTERNAL_ERROR

Indicates an internal error.

If your application receives this message you found a bug. Please report this issue to our support!

- **error**: Optional detailed error message.

```
{
  "msg": "INTERNAL_ERROR",
  "error": "Unexpected condition"
}
```

5.12 INVALID

Indicates a broken JSON message.

If your application receives this message you passed a broken JSON structure to the AusweisApp2.

Please fix your JSON document and send it again!

- **error**: Detailed error message.

```
{
  "msg": "INVALID",
  "error": "unterminated string (offset: 12)"
}
```


5.13 READER

Provides information about a connected or disconnected card reader.

This message will be send by the AusweisApp2 if a card reader was added or removed to the operating system. Also if a card was inserted into a card reader or removed from a card reader.

Your application can explicitly check for card reader with *GET_READER* (page 16).

If a workflow is in progress and a card with disabled eID functionality was inserted, this message will still be sent, but the workflow will be paused until a card with enabled eID functionality is inserted.

New in version 1.16.0: Parameter **keypad** added.

- **name**: Identifier of card reader.
- **attached**: Indicates whether a card reader is connected or disconnected.
- **keypad**: Indicates whether a card reader has a keypad. The parameter is only shown when a reader is attached.
- **card**: Provides information about inserted card, otherwise null.
  - **inoperative**: True if PUK is inoperative and cannot unblock PIN, otherwise false. This can be recognized if user enters a correct PUK only. It is not possbible to read this data before a user tries to unblock the PIN.
  - **deactivated**: True if eID functionality is deactivated, otherwise false.
  - **retryCounter**: Count of possible retries for the PIN. If you enter a PIN with command *SET_PIN* (page 19) it will be decreased if PIN was incorrect.


```
{
  "msg": "READER",
  "name": "NFC",
  "attached": true,
  "keypad": false,
  "card":
          {
           "inoperative": false,
           "deactivated": false,
           "retryCounter": 3
          }
}
```


5.14 READER_LIST

Provides information about all connected card readers.

- **reader**: A list of all connected card readers. Please see message *READER* (page 30) for details.

```
{
  "msg": "READER_LIST",
  "reader":
            [
              {
               "name": "Example reader 1 [SmartCard] (1234567) 01 00",
               "attached": true,
               "keypad": true,
               "card": null
              },

              {
               "name": "NFC",
               "attached": true,
               "keypad": false,
               "card":
                       {
                        "inoperative": false,
                        "deactivated": false,
                        "retryCounter": 3
                       }
              }
            ]
}
```


5.15 UNKNOWN_COMMAND

Indicates that the command type is unknown.

If your application receives this message you passed a wrong command to the AusweisApp2.

Please fix your command and send it again!

Be aware of case sensitive names in *Commands* (page 16).

- **error**: Name of the unknown command.


```
{
  "msg": "UNKNOWN_COMMAND",
  "error": "get_INFo"
}
```


6 Workflow

This section shows some possible workflows as an example communication between your application and the AusweisApp2.

The JSON structure can be identified by parameter **cmd** or parameter **msg** as described in section *Commands* (page 16) and section *Messages* (page 21).

- **cmd**: Commands are sent by your application.
- **msg**: Messages are sent by the AusweisApp2.

6.1 Minimal successful authentication

The following messages and commands are the minimal iterations of a successful authentication.

We assume that the user already inserted a card into the connected card reader.

```
{"cmd": "RUN_AUTH", "tcTokenURL": "https://test.governikus-eid.de/DEMO"}

{"msg": "AUTH"}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["FamilyName","GivenNames",
↪"DocumentType"],"optional":["GivenNames"],"required":["FamilyName",
↪"DocumentType"]}}

{"cmd": "ACCEPT"}

{"msg": "ENTER_PIN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":3},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_PIN", "value": "123456"}

{"msg": "AUTH","result": {"major":"http://www.bsi.bund.de/ecard/api/1.1/
↪resultmajor#ok"},"url":"https://test.governikus-eid.de/DEMO/?refID=123456
↪"}
```

6.2 Successful authentication with CAN

The following messages and commands show possible iterations if the user enters an incorrect PIN and CAN twice before entering the correct CAN and PIN.

We assume that the user did not insert a card into the connected card reader.

```
{"cmd": "RUN_AUTH", "tcTokenURL": "https://test.governikus-eid.de/DEMO"}

{"msg": "AUTH"}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["DocumentType"],"optional
↪":[],"required":["DocumentType"]}}

{"cmd": "ACCEPT"}

{"msg": "INSERT_CARD"}

{"msg": "ENTER_PIN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":3},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_PIN", "value": "000000"}

{"msg": "ENTER_PIN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":2},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_PIN", "value": "000001"}
```

(continues on next page)

(continued from previous page)

```
{"msg": "ENTER_CAN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":1},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_CAN", "value": "000000"}

{"msg": "ENTER_CAN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":1},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_CAN", "value": "654321"}

{"msg": "ENTER_PIN", "reader": {"attached":true,"card":{"inoperative
↪":false,"deactivated":false,"retryCounter":1},"keypad":false,"name":"NFC
↪"}}

{"cmd": "SET_PIN", "value": "123456"}

{"msg": "AUTH","result": {"major":"http://www.bsi.bund.de/ecard/api/1.1/
↪resultmajor#ok"},"url":"https://test.governikus-eid.de/DEMO/?refID=123456
↪"}
```

6.3 Cancelled authentication

The following messages and commands show possible iterations if the user cancels the authentication.

We assume that the user did not connect the card reader.

```
{"cmd": "RUN_AUTH", "tcTokenURL": "https://test.governikus-eid.de/DEMO"}

{"msg": "AUTH"}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["DocumentType"],"optional
↪":[],"required":["DocumentType"]}}

{"cmd": "CANCEL"}

{"msg": "AUTH", "result": {"description":"The process was cancelled by the␣
↪user.","language":"en","major":"http://www.bsi.bund.de/ecard/api/1.1/
↪resultmajor#error","message":"The process was cancelled by the user.",
↪"minor":"http://www.bsi.bund.de/ecard/api/1.1/resultminor/sal
↪#cancellationByUser"},"url":"https://test.governikus-eid.de/DEMO/?
↪errID=123456"}
```

6.4 Set some access rights

The following messages and commands show possible iterations if the user disables and enables an access right.

We assume that the user did not connect the card reader.

```
{"cmd": "RUN_AUTH", "tcTokenURL": "https://test.governikus-eid.de/DEMO"}
```

(continues on next page)

(continued from previous page)


```
{"msg": "AUTH"}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["FamilyName","GivenNames",
↪"DocumentType"],"optional":["GivenNames"],"required":["FamilyName",
↪"DocumentType"]}}

{"cmd": "SET_ACCESS_RIGHTS", "chat": []}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["FamilyName","DocumentType
↪"],"optional":["GivenNames"],"required":["FamilyName","DocumentType"]}}

{"cmd": "SET_ACCESS_RIGHTS", "chat": ["GivenNames"]}

{"msg": "ACCESS_RIGHTS", "chat": {"effective":["FamilyName","GivenNames",
↪"DocumentType"],"optional":["GivenNames"],"required":["FamilyName",
↪"DocumentType"]}}

{"cmd": "CANCEL"}

{"msg": "AUTH", "result": {"description":"The process was cancelled by the␣
↪user.","language":"en","major":"http://www.bsi.bund.de/ecard/api/1.1/
↪resultmajor#error","message":"The process was cancelled by the user.",
↪"minor":"http://www.bsi.bund.de/ecard/api/1.1/resultminor/sal
↪#cancellationByUser"},"url":"https://test.governikus-eid.de/DEMO/?
↪errID=123456"}
```