Step 1: Download the ADP Core Libraries

The ADP Core Library is a configurable library that includes access to all of the APIs supported by ADP. Rather than downloading individual API Product based libraries, configuration files based on your Consumer Application's allowed APIs are used to configure the core libraries. This eliminates the need to wait for additional libraries to become available from ADP.

The ADP Core Library supports several interfaces to get data, create and send events, and receive event notifications. These interfaces are described in detail below.

The Core Library can be obtained from github at https://github.com/adplabs/adp-core-node.

Note that in order to use the ADP Core Library it is necessary to have a registered Consumer Application with ADP. If you do not yet have a Consumer Application, please contact your ADP representative.

Step 2: Identify and place the config file

In order to use the ADP Client Core Library, it is necessary to create a Consumer Application with ADP. When a Consumer Application is created, scopes are defined that determine which specific APIs the application has access to. Once the Consumer Application is created you can request an ADP Core Library configuration file.

The configuration file is a compressed file containing an application configuration file, and schema definitions for the APIs that the Consumer Application is allowed to use. This configuration bundle is used to configure the Core Library in order to access the APIs allowed by the Consumer Application.

Place the configuration file in a location that will be accessible to your application.
A readme file will accompany the configuration file. The readme file contains a listing of the available methods as well as the list of attributes each method supports or requires.

Step 3: Create a Connection Object

The ADP Client Connection Library is intended to simplify and aid the process of authenticating, authorizing and connecting to the ADP API Gateway. The library supports OAuth2 Authorization Code and Client Credentials flows.

The library has been updated to add the ability to reconnect, and simplifies the configuration process for a connection by using JSON objects to configure the connection.

The code example below shows how to create a connection based on the updated Connection Library.

```
const adpConnection = require('adp-connection');

const init = {
     apiUrl: 'https://api.adp.com',
     tokenUrl: 'https://api.adp.com/auth/oauth/v2/token',
     authorizationUrl: 'https://accounts.adp.com/auth/oauth/v2/authorize',
     sslCertPath: 'certs/apiclient_iat.pem',
     sslKeyPath: 'certs/apiclient_iat.key',
     granttype: 'client_credentials',
     clientId: ‘xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx’,
     clientSecret: ‘xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx’
};

const connection = adpConnection.createConnection(init);
connection.connect((err) => {
     console.log(‘Received: ', err, connection.accessToken);
});
```

Step 4: Create a Consumer Application Instance

The Consumer Application Instance object is the main object used for getting and sending data and events. Consumer Application Instances are created using the consumerApplicationInstance() method in the core library. This method is called with a connection and the location of a configuration file, and returns a Consumer Application Instance object, which can then be used for subsequent calls to the library.

The code example below shows how to create a Consumer Application Instance after the configuration file has been received and a connection object has been created.

```
const adpCore = require('adp-core');

const app = adpCore.consumerApplicationInstance(connection, './config.zip');
```

Step 5: Getting Data with the ADP Core Library

The getData() method of the Consumer Application Instance is used to read data from ADP.

The getData() method takes a method name and an optional parameter object as arguments. The list of available method names and parameters can be found in the readme file that accompanies the configuration file.

The data will be returned as a JSON formatted object. The structure of the JSON object is the same as that returned by a normal REST API call.

The code example below shows how to retrieve user info data for a particular associate after the Consumer Application Instance has been created.

```
app.getData(
    ‘core.v1.associates.associateoid.user.info’,
    { associateoid: ‘FFEEDDAABBCC0011’},
    (err, response) => {
        console.log(‘Received: ', err, response);
    }
);
```

Step 6: Create Event Objects

In order to update data using the Core Library, it is necessary to create and send JSON formatted event payload objects. Since the structure and content of event objects varies between event types, a createEvent() method is available which creates a JSON payload object.

The createEvent() method takes an event name as an argument. The list of available event can be found in the readme file that accompanies the configuration file.

The resulting payload object will contain placeholders that can be modified as required. Examples of placeholders are DELETE_STRING and DELETE_INTEGER.

An excerpt of a payload object is shown below.

```
"transform": {
    "worker": {
     "person": {
       "legalName": {
          "givenName": "DELETE_STRING",
         "middleName": "DELETE_STRING",
          "familyName1": "DELETE_STRING",
         "familyName2": "DELETE_STRING",
         "formattedName": "DELETE_STRING",
```

The resulting JSON object can be updated as required and then used with the saveEvent() method. The JSON object will also contain an ‘eventID’ attribute that should not be removed or modified as it is used by the Core Library to map to the use of createEvent() is not required, though it does guarantee that the resulting payload object is correctly structured.

Note that it is not necessary to remove the placeholder entries prior to calling the saveEvent() method, as they will be stripped out when setting the data.

The code example below shows how to create a legal name change event payload object after the Consumer Application Instance has been created.

```
app.createEvent(
    {methodName: ‘events.hr.v1.worker.legal.name.change’},
    (err, eventPayload) => {
        console.log(‘Received: ', err, eventPayload);
    }
);
```

Step 7: Saving Data with the ADP Core Library

To save data it is necessary to create an event payload object, update it's contents, and then save it. The saveEvent() method takes an event object as an argument and sends it to the appropriate ADP endpoint.

Prior to sending the event, the event payload object will be validated prior to sending the event. If any errors are found during validation they will be returned as an array of error messages. An excerpt of a validation failure object is shown below.

```
[ ValidationException {
    message: 'Value is not optional and must be set.',
    pathToError: 'events/0/data/transform/worker/person/legalName/givenName',
    description: 'Non-optional value',
    context: '' },
  ValidationException {
    message: 'Value is not within acceptable range between 1 and 64',
    pathToError: 'events/0/data/transform/worker/person/legalName/givenName',
    description: 'Value length',
    context: ''
} ]
```

After processing the event, a response data will be returned that contains a JSON object with the results of the call. The structure of the JSON object is the same as that returned by a normal REST API call to post an event.

The following code example below shows how to update the last name for a worker, after the Consumer Application Instance and legal name change payload has been created.

```
event.events[0].data.eventContext.worker.associateOID = 'FFEEDDAABBCC0011';
event.events[0].data.transform.worker.person.legalName.givenName = 'Thomas';
event.events[0].data.transform.worker.person.legalName.familyName1 = 'Anderson';

app.saveEvent( payload, (err, response) => {
    console.log( "Received: ", err, response );
});
```

Step 8: Get Next Event with the ADP Core Library

The getNextEvent() method is used to retrieve events that are waiting on the event notifications queue. The getNextEvent() method will return the next available event as a JSON event object. If an event is not available it will return an empty object.

After the event has been successfully retrieved, it will be automatically deleted so that subsequent calls to getNextEvent() will return new events (or an empty object).

The structure of the resulting JSON event object is the same as that returned by a normal REST API call.

The code example below shows how to retrieve event notifications after the Consumer Application Instance has been created.

```
app.getNextEvent( (err, body, headers ) => {
    console.log( 'Received: ', err, body, headers );
});
```