# Enhanced WebRTC Migration Guide (prerelease to v1.0)

This document describes how to migrate an existing app to the v1.0 release of the Enhanced WebRTC SDK.

## Audience

If you have created an application using the prerelease version of the Enhanced WebRTC SDK, using these features of the SDK:

* Node.js
* DHS app that shipped with the SDK
* Methods in the ATT.rtc.dhs namespace

You will need to migrate your application as described below.

## Migration Overview

### DHS Services

The prerelease version of the Enhanced WebRTC SDK required a dedicated node.js server app, in addition to the app server that implemented the pages using the SDK. That server was called the Developer Hosted Server (DHS), and it was included with the SDK release. DHS provided a secure intermediary between the browser client and the AT&T WebRTC services.

App

DHS

WebRTC Client Library

Client

App Code

SDK Code

In v1.0 of the SDK, the functionality of the DHS is provided as a loadable module: att-dhs. The app developer can choose whether to expose this functionality as a separate server, like the prerelease SDK; or to load the module into their existing app server.

App

DHS

App

att-dhs

att-dhs

OR

App Code

SDK Code

Client

Client

WebRTC Client Library

WebRTC Client Library

NPM Module

In addition to replacing the standalone DHS server with the new att-dhs module, the app developer will need to implement endpoints exposing required services to the browser, and will need to implement the browser client code to call those endpoints. Server and client code snippets are provided for all DHS services.

### DHS Configuration

The prerelease version of the SDK kept common shared configuration values in the package.json file of the DHS server. In v1 of the SDK, these values must be passed in to the configure method of the att-dhs module.

There is a more detailed example of this migration in the configure section below.

## Migration Example

### Here is an example demonstrating how one of the DHS methods could be migrated, with example code from the sample that ships with the SDK.

### Your Client Code, Before Migration

In the prerelease SDK you would obtain an access token in the client like this:

ATT.rtc.dhs.createAccessToken({

app\_scope: appScope,

auth\_code: authCode,

success: success,

error: error

});

Internally this would call the standalone DHS server, which would do the secure operations necessary to obtain the token, and return it to the client API above.

### Your Server Code, After Migration

Instead of running the standalone DHS server, you will import the att-dhs module into your Node.js app:

var dhs = require('att-dhs');

 The dhs object implements a createAccessToken method. You must expose this functionality to the browser client; the sample application that ships with the SDK does this by exposing a POST endpoint /oauth/token as follows:

var oauth = express.Router();

oauth.post('/tokens', function (req, res) {

console.log('Got token request');

var app\_scope = req.body.app\_scope,

auth\_code = req.body.auth\_code;

console.log('App scope: %s', req.body.app\_scope);

console.log('Auth code: %s', req.body.auth\_code);

console.log('Creating access token');

try {

dhs.createAccessToken({

app\_scope: app\_scope,

auth\_code: auth\_code,

success: function (response) {

console.log('Success in creating access token: %s', response);

res.json(200, response);

},

error: function (error) {

console.log('Error in creating access token: %s', error);

res.json(400, error);

}

});

} catch (error) {

console.log('Error: %s', error.message);

res.json(400, {

error: error.message

});

}

});

var app = express();

app.use('/oauth', oauth);

### Your Client Code, After Migration

In the browser client, instead of calling the SDK method ATT.rtc.dhs.createAccessToken, you make an AJAX call to the new endpoint on the server:

var xhr = new XMLHttpRequest();

xhr.open("POST", "https://myapp.com/oauth/tokens");

xhr.setRequestHeader("Content-Type", "application/json");

xhr.setRequestHeader("Accept", "application/json");

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

useAccessToken(JSON.parse(xhr.responseText).access\_token);

} else {

reportError(xhr.responseText);

}

}

xhr.send(

JSON.stringify({app\_scope: appScope, auth\_code: authCode})

);

## Migration Scope

This section enumerates the APIs from the prerelease version of the Enhanced WebRTC SDK that must be migrated as described above.

Please refer to the [sample application](https://github.com/attdevsupport/ewebrtc-sdk/tree/master/node-sample) to see how [the server endpoints](https://github.com/attdevsupport/ewebrtc-sdk/blob/master/node-sample/routes/att.js) can be implemented.

### configure

In the prerelease version of the SDK, common configuration values were loaded from the DHS app’s package.json file:

"app\_key": "xxxxxxxxxxxxxxxxxxxxxxxxx",

"app\_secret": "xxxxxxxxxxxxxxxxxxxxxxxxx",

(Note that the two values above are the only *required* entries.)

In the v1 of the SDK, you instead pass these values as object properties to the configure method on the att-dhs module:

var dhs = require('att-dhs');

dhs.configure({

app\_key: "xxxxxxxxxxxxxxxxxxxxxxxxx",

app\_secret: "xxxxxxxxxxxxxxxxxxxxxxxxx"

});

### createAccessToken

This method obtains an OAuth token that can be used to authorize operations that do not require explicit user consent. The specific operations authorized depends on the scope parameter to this method. For example, to create an E911 ID associated with a particular address, you will need a token generated for the “E911” scope.

The sample app uses the endpoint /oauth/tokens to expose this method to the client.

Please refer to the Your Client Code, After Migration section above for a client code snippet.

### createE911Id

Some Enhanced WebRTC operations require a registered E911 physical address. In order to associate that address with the operation, you must obtain an E911 ID that represents the physical address. This method gets an E911 ID for a specified physical address.

The sample app uses the endpoint /e911ids to expose this method to the client.

The following client code snippet can be used to access that endpoint on the sample app:

var xhr = new XMLHttpRequest();

xhr.open("POST", "https://myapp.com/e911ids");

xhr.setRequestHeader("Content-Type", "application/json");

xhr.setRequestHeader("Accept", "application/json");

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

useE911Id(JSON.parse(xhr.responseText));

} else {

reportError(xhr.responseText);

}

}

xhr.send(JSON.stringify({

token: "your-e911-auth-token-here",

address: {

first\_name: "John",

last\_name: "Doe",

house\_number: "1234",

street: "Franklin",

street\_suffix: "Avenue",

city: "Kirkland",

state: "WA",

zip: "98033"

}

}));

### getAuthorizeUrl

Some Enhanced WebRTC operations require explicit user consent. In order to obtain that consent, the application must call the getAuthorizeUrl method to get a link to the AT&T authorization page, and redirect or otherwise navigate the user to that page.

The sample application uses the endpoint /oauth/authorize to redirect the client.

The following client code snippet can be used to navigate to the AT&T authorization page:

window.location.href = "https://myapp.com/oauth/authorize";

### getConfiguration

The Enhanced WebRTC client library must be initialized with the app’s common configuration values. The getConfiguration method on the att-dhs module can be used to obtain these values.

The sample application uses the endpoint /config to return these values to the client.

The following client code snippet can be used to access that endpoint on the sample app:

var xhr = new XMLHttpRequest();

xhr.open("GET", "https://myapp.com/config");

xhr.setRequestHeader("Accept", "application/json");

xhr.onreadystatechange = function() {

if (xhr.readyState == 4) {

if (xhr.status == 200) {

ATT.rtc.configure(JSON.parse(xhr.responseText));

} else {

reportError(xhr.responseText);

}

}

xhr.send();