EXERCISE 11A - Get Set to Use SAP Cloud SDK for JavaScript

SAP Partner Workshop	**15 min**

Description

In this exercise, you'll learn how

- How to extend a scaffolded application by another route
- How to call the Business Partner Service of SAP S/4HANA Cloud using SAP Cloud SDK for JavaScript

Target group

- Developers
- People interested in learning about S/4HANA Cloud SDK for JavaScript.

Goal

The goal of this exercise is to learn the fundamentals of the SAP Cloud SDK for JavaScript and integrate with an SAP S/4HANA Cloud system.

Prerequisites

- Node.js, npm, SAP Cloud Platform Trial Account.
- Exercise 10A & 10B must be completed.

Steps

Step 1: Set up a local mock server or get access to the API Business Hub sandbox (optional)

Note: If you have access to an SAP S/4HANA Cloud system with a technical user, you can skip this part.

In order to make a call to an OData service, there needs to be a service to call. You can setup a local mock server that mimics the business partner and a custom service by following the instructions here. This mock server does not support all the features of the actual OData services, but it suffices to try it out locally.

Once it is up and running you should see the list of services at http://localhost:3000/.

Alternatively, many APIs can also be tested using the sandbox of the SAP API Business Hub. To use the sandbox, you need an an API key. Go to https://api.sap.com and click “Log On” in the top right corner. If you do not have an account, you will need to register first. When you are logged in, click on “Hi ” in the top right corner and then click on “Preferences” in the dropdown menu that just opened. On the preferences page, click on “Show API Key”.

Step 2: Get familiar with the project

Initially, the app only contains the index and hello-world routes. We will add another route for business-parters that will simply list all available business partners.

First, create a new file business-partner-route.ts in the src/ directory and add an implementation for this route, like so:

```
import { Request, Response } from 'express';

export function businessPartners(req: Request, res: Response) {
  res.status(200).send('We will implement this in a minute');
}
```

The businessPartners function is a callback function that we will register for a specific route. It writes the status 200 and a placeholder message to the response.

Then, add this route to the routes of your application in application.ts (see // add the following line to your code):

```
private routes(): void {
  const router = express.Router();
  router.get('/', indexRoute);
  router.get('/hello', helloWorld);
  // add the following line to your code
  router.get('/business-partners', businessPartners);
  this.app.use('/', router);
}
```

Should your editor not offer you to automatically add the correct import, add the following line to your import statements:

```
import { businessPartners } from './business-partner-route';
```

You can start your application by running npm run start:local. Now, calling http://localhost:8080/business-partners should return our placeholder string.

Step 3: Import Service entities

In order to use the SAP Cloud SDK for JavaScript to make a call to an OData service add the virtual data model (VDM) for this service to your dependencies. For this tutorial we are using the VDM for the business partner service. Install it with the following:

```
npm install @sap/cloud-sdk-vdm-business-partner-service
```

Import the entity you want to make a call to into your application. In this tutorial we are importing the business partner entity of the business partner service. Add the following line to the top of the business-partner-route.ts.

Now the BusinessPartner entity is available for you to be used.

Side-note: The SAP Cloud SDK for JavaScript offers packages for each OData service exposed by SAP S/4HANA Cloud. You can find a list of these services in the SAP API Business Hub and a list of the corresponding packages in our documentation.

Step 4: Execute an Odata request

In the business-partner-route create a function getAllBusinessPartners and implement it as follows:

```
function getAllBusinessPartners(): Promise<BusinessPartner[]> {
  return BusinessPartner.requestBuilder()
    .getAll()
    .execute({ url: 'http://localhost:3000'
  });
}
```

- In line 1, we are creating a request builder for the business partner entity.
- Line 2 indicates, that we want to create a request to get all the business partners.
- Line 3 ff. takes care of the execution and sends a request to a url based on the given destination url.

In the code snippet above we assume that you have a mock server running locally. If you are using an actual SAP S/4HANA Cloud system, you can replace the third line with a different destination configuration:

```
.execute({
  url: '<URI of your SAP S/4HANA Cloud System>',
  username: '<USERNAME>',
  password: '<PASSWORD>'
})
```

To use the SAP API Business Hub sandbox for your requests, you will need to pass the API key to the VDM requests using the withCustomHeaders method, and you will need to add the correct URL to your destinations. Checkout the following example:

```
return BusinessPartner.requestBuilder()
  .getAll()
  .withCustomHeaders({ APIKey: '<YOUR-API-KEY>'})
  .execute({
    url: 'https://sandbox.api.sap.com/s4hanacloud'
  });
}
```

As network requests are asynchronous by nature, the return value of this function is a Promise to a list of Business Partners (Promise<BusinessPartner[]>).

Let's add the execution of this request to the callback for our business-partners route by making the callback **async** and writing the resolved return value of the getAllBusinessPartners function to the response:

```
export async function businessPartners(req: Request, res: Response) {
  getAllBusinessPartners()
    .then(businessPartners => res.status(200).send(businessPartners))
```

```
    .catch(error => res.status(500).send(error));
}
```

Here is what your business-partner-route.ts should look like, if you are using the mock server:

```
import { BusinessPartner } from "@sap/cloud-sdk-vdm-business-partner-service"; import { Request, Response } from "express";

export async function businessPartners(req: Request, res: Response) {
  getAllBusinessPartners()
    .then(businessPartners => res.status(200).send(businessPartners))
    .catch(error => res.status(500).send(error));
}

function getAllBusinessPartners(): Promise<BusinessPartner[]> {
  return BusinessPartner.requestBuilder()
    .getAll()
    .execute({ url: 'http://localhost:3000' });
}
```

Now restart your server and reload the http://localhost:8080/business-partners url to retrieve a list of business partners.

Congratulations, you just made your first call with the SAP Cloud SDK!

Step 5: Manage Destinations Centrally

In order to not repeat your destination configuration for every request execution, you can set a destinations environment variable to manage your destinations. If you prefer, you can set system wide environment variables. However, we will show you how to set them non-invasively for one project only.

We will use the dotenv npm module to facilitate setting environment variables on a node process.

```
# install dotenv as a devDependency
npm install --save-dev dotenv
```

Then create a .env file in the root directory of your project and define your a destinations environment variable as follows:

```
destinations=[{"name": "<DESTINATIONNAME>", "url": "<URL to your system>", "username": "<USERNAME>", "password": "<PASSWORD>"}]
```

This is what it would look like for the mock server:

destinations=[{"name": "MockServer", "url": "http://localhost:3000"}]

Now to reference a destination in the request execution, simply replace the `url` with a `destinationName` - `MockServer` in our example:

```
function getAllBusinessPartners(): Promise<BusinessPartner[]> {
  return BusinessPartner.requestBuilder()
    .getAll()
    .execute({ destinationName: 'MockServer'
  });
}
```

Note, that every environment variable in the `.env` file has to be defined *on one line*. You can add more destinations to the array.

In order to register the `.env` file in your node process, adjust the `"start:local"` script in the `package.json` as follows:

```
"start:local": "npx ts-node -r dotenv/config src/"
```

Now, when you execute `npm run start:local` and call `localhost:8080/business-partners`, you can also use your remote destination locally.

You can proceed to exercise 11B.