Consuming Services

▶ *This guide is available for Node.js and Java.*

Table of Contents

Introduction

If you want to use data from other services or you want to split your application into multiple microservices, you need a connection between those services. We call them **remote services**. As everything in CAP is a service, remote services are modeled the same way as internal services — using CDS.

CAP supports service consumption with dedicated APIs to import service definitions, query remote services, mash up services, and work locally as much as possible.

Feature Overview

For outbound remote service consumption, the following features are supported:

- OData V4
- OData V2 (Deprecated)
- Querying API
- Projections on remote services

Tutorials and Examples

Example	Description
Capire Bookstore (Fiori)	Example, Node.js, CAP-to-CAP
Example Application (Node.js)	Complete application from the end-to-end Tutorial
Example Application (Java)	Complete application from the end-to-end Tutorial
Incident Management (Node.js)	Using a mock server or S/4 on Cloud Foundry or Kyma

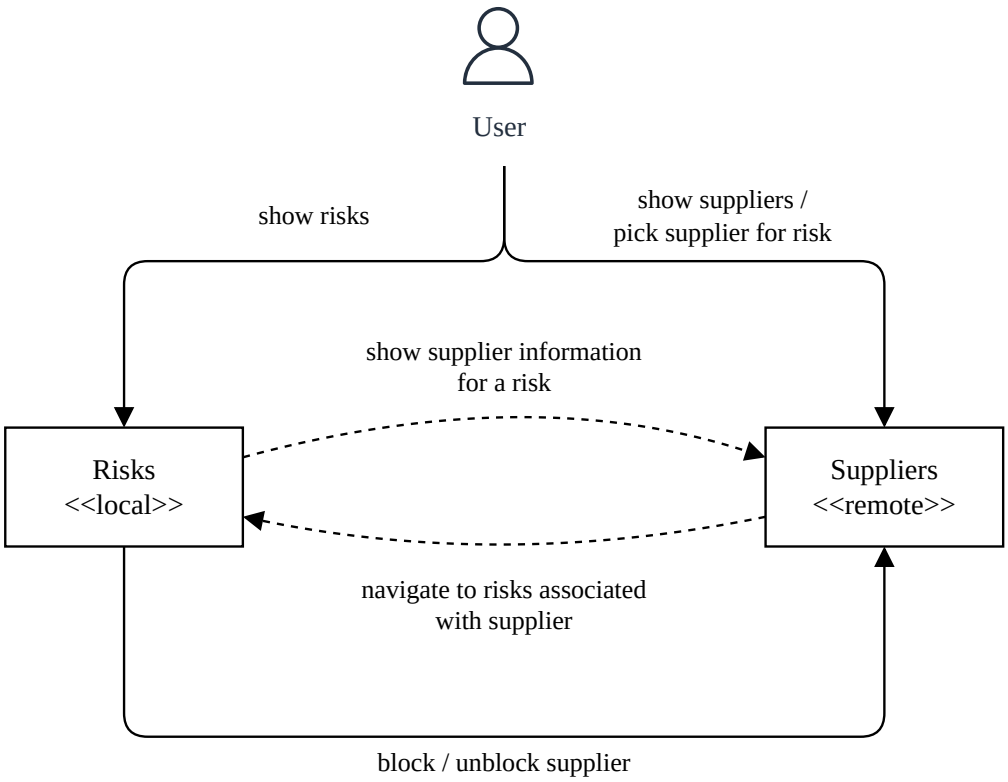
Define Scenario

Before you start your implementation, you should define your scenario. Answering the following questions gets you started:

- What services (remote/CAP) are involved?
- How do they interact?
- What needs to be displayed on the UI?

You have all your answers and know your scenario, go on reading about external service APIs, getting an API definition from the SAP Business Accelerator Hub or from a CAP project, and importing an API definition to your project.

Sample Scenario from End-to-End Tutorial



> ***User Story***
>
> A company wants to ensure that goods are only sourced from suppliers with acceptable risks. There shall be a software system, that allows a clerk to maintain risks for suppliers and their mitigations. The system shall block the supplier used if risks can't be mitigated.

The application is an extension for SAP S/4HANA. It deals with *risks* and *mitigations* that are local entities in the application and *suppliers* that are stored in SAP S/4HANA Cloud. The application helps to reduce risks associated with suppliers by automatically blocking suppliers with a high risk using a remote API Call.

Integrate

The user picks a supplier from the list. That list is coming from the remote system and is exposed by the CAP application. Then the user does a risk assessment. Additional supplier data, like name and blocked status, should be displayed on the UI as well, by integrating the remote supplier service into the local risk service.

Extend

It should be also possible to search for suppliers and show the associated risks by extending the remote supplier service with the local risk service and its risks.

> **New scenario: Incident Management**

If you want to learn about this topic based on the Incident Management sample, you can follow the BTP Developer's Guide repository .

Install Dependencies

```sh
npm add @sap-cloud-sdk/http-client@4.x @sap-cloud-sdk/connectivity@4.x @sap-c
```

Get and Import an External Service API

To communicate to remote services, CAP needs to know their definitions. Having the definitions in your project allows you to mock them during design time.

These definitions are usually made available by the service provider. As they aren't defined within your application but imported from outside, they're called *external* service APIs in CAP. Service APIs can be provided in different formats. Currently, *EDMX* files for OData V2 and V4 are supported.

From SAP Business Accelerator Hub

The SAP Business Accelerator Hub provides many relevant APIs from SAP. You can download API specifications in different formats. If available, use the EDMX format. The EDMX format describes OData interfaces.

To download the Business Partner API (A2X) from SAP S/4HANA Cloud , go to section **API Resources**, select **API Specification**, and download the **EDMX** file.

For a Remote CAP Service

We recommend using EDMX as exchange format. Export a service API to EDMX:

Mac/Linux Windows Powershell

```sh
cds compile srv -s OrdersService -2 edmx > OrdersService.edmx
```

↳ *You can try it with the orders sample in **capire/orders**.*

By default, CAP works with OData V4 and the EDMX export is in this protocol version as well. The `cds compile` command offers options for other OData versions and flavors, call `cds help compile` for more information.

> **Don't just copy the CDS file for a remote CAP service**
>
> Simply copying CDS files from a different application comes with the following issues:
>
> - The effective service API depends on the used protocol.
> - CDS files often use includes, which can't be resolved anymore.
> - CAP creates unneeded database tables and views for all entities in the file.

Import API Definition

Import the API to your project using `cds import` .

```sh
cds import <input_file> --as cds
```

> `<input_file>` can be an EDMX (OData V2, OData V4), OpenAPI or AsyncAPI file.

Option	Description
`--as cds`	The import creates a CDS file (for example *API_BUSINESS_PARTNER.cds*) instead of a CSN file.

This adds the API in CDS format to the *srv/external* folder and also copies the input file into that folder.

Further, it adds the API as an external service to your *package.json*. You use this declaration later to connect to the remote service using a destination.

```json
"cds": {
    "requires": {
        "API_BUSINESS_PARTNER": {
            "kind": "odata-v2",
            "model": "srv/external/API_BUSINESS_PARTNER"
        }
    }
}
```

▶ *Options and flags in .cdsrc.json*

When importing the specification files, the `kind` is set according to the following mapping:

Imported Format	Used `kind`
OData V2	`odata-v2`
OData V4	`odata` (alias for `odata-v4`)
OpenAPI	`rest`
AsyncAPI	`odata`

↳ *Learn more about type mappings from OData to CDS and vice versa.*

> **TIP**
>
> Always use OData V4 (`odata`) when calling another CAP service.

> **Limitations**
>
> Not all features of OData, OpenAPI, or AsyncAPI are supported in CAP which may lead to the rejection of the imported model by the CDS compiler or may result in a different API when rendered by CAP. Known limitations are cyclic type references and inheritance.

Local Mocking

When developing your application, you can mock the remote service.

Add Mock Data

As for any other CAP service, you can add mocking data.

The CSV file needs to be added to the *srv/external/data* folder.

API_BUSINESS_PARTNER-A_BusinessPartner.csv

```csv
BusinessPartner,BusinessPartnerFullName,BusinessPartnerIsBlocked
1004155,Williams Electric Drives,false
1004161,Smith Batteries Ltd,false
1004100,Johnson Automotive Supplies,true
```

↳ *Find this source in the end-to-end tutorial*

Run Local with Mocks

Start your project with the imported service definition.

```sh
cds watch
```

The service is automatically mocked, as you can see in the log output on server start.

```log
...

[cds] - model loaded from 8 file(s):

  ...
  ./srv/external/API_BUSINESS_PARTNER.cds
  ...

[cds] - connect using bindings from: { registry: '~/.cds-services.json' }
[cds] - connect to db > sqlite { database: ':memory:' }
 > filling sap.ui.riskmanagement.Mitigations from ./db/data/sap.ui.riskmanage
 > filling sap.ui.riskmanagement.Risks from ./db/data/sap.ui.riskmanagement-R
 > filling API_BUSINESS_PARTNER.A_BusinessPartner from ./srv/external/data/AP
/> successfully deployed to sqlite in-memory db
```

```
[cds] - serving RiskService { at: '/service/risk', impl: './srv/risk-service.
[cds] - mocking API_BUSINESS_PARTNER { at: '/api-business-partner' }

[cds] - launched in: 1.104s
[cds] - server listening on { url: 'http://localhost:4004' }
[ terminate with ^C ]
```

↳ *If you want to run with a mock server in the cloud, try the BTP Developer's Guide.*

Mock Associations

You can't get data from associations of a mocked service out of the box.

The associations of imported services lack information how to look up the associated records. This missing relation is expressed with an empty key definition at the end of the association declaration in the CDS model (`{ }`).

srv/external/API_BUSINESS_PARTNER.cds

```cds
entity API_BUSINESS_PARTNER.A_BusinessPartner {
  key BusinessPartner : LargeString;
  BusinessPartnerFullName : LargeString;
  BusinessPartnerType : LargeString;

  ...

  to_BusinessPartnerAddress :
    Association to many API_BUSINESS_PARTNER.A_BusinessPartnerAddress {  };
};

entity API_BUSINESS_PARTNER.A_BusinessPartnerAddress {
  key BusinessPartner : String(10);
  key AddressID : String(10);

  ...
};
```

To mock an association, you have to modify the imported file. Before doing any modifications, create a local copy and add it to your source code management system.

```sh
cp srv/external/API_BUSINESS_PARTNER.cds srv/external/API_BUSINESS_PARTNER-or
git add srv/external/API_BUSINESS_PARTNER-orig.cds
...
```

Import the CDS file again, just using a different name:

```sh
cds import ~/Downloads/API_BUSINESS_PARTNER.edmx --keep-namespace \
    --as cds --out srv/external/API_BUSINESS_PARTNER-new.cds
```

Add an *on* condition to express the relation:

srv/external/API_BUSINESS_PARTNER-new.cds

```cds
entity API_BUSINESS_PARTNER.A_BusinessPartner {
  // ...
  to_BusinessPartnerAddress :
      Association to many API_BUSINESS_PARTNER.A_BusinessPartnerAddress
      on to_BusinessPartnerAddress.BusinessPartner = BusinessPartner;
};
```

Don't add any keys or remove empty keys, which would change it to a managed association. Added fields aren't known in the service and lead to runtime errors.

Use a 3-way merge tool to take over your modifications, check it and overwrite the previous unmodified file with the newly imported file:

```sh
git merge-file API_BUSINESS_PARTNER.cds \
               API_BUSINESS_PARTNER-orig.cds \
               API_BUSINESS_PARTNER-new.cds
mv API_BUSINESS_PARTNER-new.cds API_BUSINESS_PARTNER-orig.cds
```

To prevent accidental loss of modifications, the *cds import --as cds* command refuses to overwrite modified files based on a "checksum" that is included in the file.

Mock Remote Service as OData Service (Node.js)

As shown previously you can run one process including a mocked external service. However, this mock doesn't behave like a real external service. The communication

happens in-process and doesn't use HTTP or OData. For a more realistic testing, let the mocked service run in a separate process.

Start the CAP application with the mocked remote service only:

```sh
cds mock API_BUSINESS_PARTNER
```

If the startup is completed, run *`cds watch`* in the same project from a **different** terminal:

```sh
cds watch
```

CAP tracks locally running services. The mocked service *`API_BUSINESS_PARTNER`* is registered in file *~/.cds-services.json*. *`cds watch`* searches for running services in that file and connects to them.

Node.js only supports *OData V4* protocol and so does the mocked service. There might still be some differences to the real remote service if it uses a different protocol, but it's much closer to it than using only one instance. In the console output, you can also easily see how the communication between the two processes happens.

Execute Queries

You can send requests to remote services using CAP's powerful querying API.

Execute Queries with Node.js

Connect to the service before sending a request, as usual in CAP:

```js
const bupa = await cds.connect.to('API_BUSINESS_PARTNER');
```

Then execute your queries using the Querying API:

```js
const { A_BusinessPartner } = bupa.entities;
const result = await bupa.run(SELECT(A_BusinessPartner).limit(100));
```

We recommend limiting the result set and avoid the download of large data sets in a single request. You can `limit` the result as in the example: `.limit(100)`.

Many features of the querying API are supported for OData services. For example, you can resolve associations like this:

```js
const { A_BusinessPartner } = bupa.entities;
const result = await bupa.run(SELECT.from(A_BusinessPartner, bp => {
    bp('BusinessPartner'),
    bp.to_BusinessPartnerAddress(addresses => {
      addresses('*')
    })
  }).limit(100));
```

↳ *Learn more about querying API examples.*

↳ *Learn more about supported querying API features.*

Model Projections

External service definitions, like generated CDS or CSN files during import, can be used as any other CDS definition, but they **don't** generate database tables and views unless they are mocked.

It's best practice to use your own "interface" to the external service and define the relevant fields in a projection in your namespace. Your implementation is then independent of the remote service implementation and you request only the information that you require.

```cds
using {  API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAI

entity Suppliers as projection on bupa.A_BusinessPartner {
  key BusinessPartner as ID,
  BusinessPartnerFullName as fullName,
  BusinessPartnerIsBlocked as isBlocked,
}
```

As the example shows, you can use field aliases as well.

↳ *Learn more about supported features for projections.*

Execute Queries on Projections to a Remote Service

Connect to the service before sending a request, as usual in CAP:

```js
const bupa = await cds.connect.to('API_BUSINESS_PARTNER');
```

Then execute your queries:

```js
const suppliers = await bupa.run(SELECT(Suppliers).where({ID}));
```

CAP resolves projections and does the required mapping, similar to databases.

A brief explanation, based on the previous query, what CAP does:

- Resolves the `Suppliers` projection to the external service interface `API_BUSINESS_PARTNER.A_Business_Partner`.
- The **where** condition for field `ID` will be mapped to the `BusinessPartner` field of `A_BusinessPartner`.
- The result is mapped back to the `Suppliers` projection, so that values for the `BusinessPartner` field are mapped back to `ID`.

This makes it convenient to work with external services.

Building Custom Requests with Node.js

If you can't use the querying API, you can craft your own HTTP requests using `send`:

```js
bupa.send({
  method: 'PATCH',
  path: A_BusinessPartner,
  data: {
    BusinessPartner: 1004155,
    BusinessPartnerIsBlocked: true
  }
})
```

↳ *Learn more about the `send` API.*

Integrate and Extend

By creating projections on remote service entities and using associations, you can create services that combine data from your local service and remote services.

What you need to do depends on the scenarios and how your remote services should be integrated into, as well as extended by your local services.

Expose Remote Services

To expose a remote service entity, you add a projection on it to your CAP service:

```cds
using {  API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAI

extend service RiskService with {
  entity BusinessPartners as projection on bupa.A_BusinessPartner;
}
```

CAP automatically tries to delegate queries to database entities, which don't exist as you're pointing to an external service. That behavior would produce an error like this:

```xml
<error xmlns="https://docs.oasis-open.org/odata/ns/metadata">
<code>500</code>
<message>SQLITE_ERROR: no such table: RiskService_BusinessPartners in: SELECT
</error>
```

To avoid this error, you need to handle projections. Write a handler function to delegate a query to the remote service and run the incoming query on the external service.

Node.js Java

```js
module.exports = cds.service.impl(async function() {
  const bupa = await cds.connect.to('API_BUSINESS_PARTNER');

  this.on('READ', 'BusinessPartners', req => {
      return bupa.run(req.query);
```

```
  });
});
```

↳ *For Node.js, get more details in the end-to-end tutorial.*

> **WARNING**
>
> If you receive `404` errors, check if the request contains fields that don't exist in the service and start with the name of an association. `cds import` adds an empty keys declaration (`{ }`) to each association. Without this declaration, foreign keys for associations are generated in the runtime model, that don't exist in the real service. To solve this problem, you need to reimport the external service definition using `cds import .`

This works when accessing the entity directly. Additional work is required to support navigation and expands from or to a remote entity.

Instead of exposing the remote service's entity unchanged, you can model your own projection. For example, you can define a subset of fields and change their names.

> **TIP**
>
> CAP does the magic that maps the incoming query, according to your projections, to the remote service and maps back the result.

cds

```
using { API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAR

extend service RiskService with {
  entity Suppliers as projection on bupa.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked
  }
}
```

js

```
module.exports = cds.service.impl(async function() {
  const bupa = await cds.connect.to('API_BUSINESS_PARTNER');

  this.on('READ', 'Suppliers', req => {
      return bupa.run(req.query);
```

```
  });
});
```

↳ *Learn more about queries on projections to remote services.*

Expose Remote Services with Associations

It's possible to expose associations of a remote service entity. You can adjust the projection for the association target and change the name of the association:

```cds
using { API_BUSINESS_PARTNER as bupa } from '../srv/external/API_BUSINESS_PAR';

extend service RiskService with {
  entity Suppliers as projection on bupa.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked,
    to_BusinessPartnerAddress as addresses: redirected to SupplierAddresses
  }

  entity SupplierAddresses as projection on bupa.A_BusinessPartnerAddress {
    BusinessPartner as bupaID,
    AddressID as ID,
    CityName as city,
    StreetName as street,
    County as county
  }
}
```

As long as the association is only resolved using expands (for example `.../risk/Suppliers?$expand=addresses`), a handler for the **source entity** is sufficient:

```js
this.on('READ', 'Suppliers', req => {
    return bupa.run(req.query);
});
```

If you need to resolve the association using navigation or request it independently from the source entity, add a handler for the **target entity** as well:

```js
this.on('READ', 'SupplierAddresses', req => {
    return bupa.run(req.query);
});
```

As usual, you can put two handlers into one handler matching both entities:

```js
this.on('READ', ['Suppliers', 'SupplierAddresses'], req => {
    return bupa.run(req.query);
});
```

Mashing up with Remote Services

You can combine local and remote services using associations. These associations need manual handling, because of their different data sources.

Integrate Remote into Local Services

Use managed associations from local entities to remote entities:

```cds
@path: 'service/risk'
service RiskService {
  entity Risks : managed {
    key ID       : UUID  @(Core.Computed : true);
    title        : String(100);
    prio         : String(5);
    supplier     : Association to Suppliers;
  }

  entity Suppliers as projection on BusinessPartner.A_BusinessPartner {
    key BusinessPartner as ID,
    BusinessPartnerFullName as fullName,
    BusinessPartnerIsBlocked as isBlocked,
  };
}
```

Extend a Remote by a Local Service

You can augment a projection with a new association, if the required fields for the on condition are present in the remote service. The use of managed associations isn't

possible, because this requires to create new fields in the remote service.

```cds
entity Suppliers as projection on bupa.A_BusinessPartner {
  key BusinessPartner as ID,
  BusinessPartnerFullName as fullName,
  BusinessPartnerIsBlocked as isBlocked,
  risks : Association to many Risks on risks.supplier.ID = ID,
};
```

Handle Mashups with Remote Services

Depending on how the service is accessed, you need to support direct requests, navigation, or expands. CAP resolves those three request types only for service entities that are served from the database. When crossing the boundary between database and remote sourced entities, you need to take care of those requests.

The list of required implementations for mashups explains the different combinations.

Handle Expands Across Local and Remote Entities

Expands add data from associated entities to the response. For example, for a risk, you want to display the suppliers name instead of just the technical ID. But this property is part of the (remote) supplier and not part of the (local) risk.

To handle expands, you need to add a handler for the main entity:

1. Check if a relevant `$expand` column is present.
2. Remove the `$expand` column from the request.
3. Get the data for the request.
4. Execute a new request for the expand.
5. Add the expand data to the returned data from the request.

Example of a CQN request with an expand:

```json
{
  "from": { "ref": [ "RiskService.Suppliers" ] },
  "columns": [
    { "ref": [ "ID" ] },
    { "ref": [ "fullName" ] },
    { "ref": [ "isBlocked" ] },
```

```
      { "ref": [ "risks" ] },
      { "expand": [
        { "ref": [ "ID" ] },
        { "ref": [ "title" ] },
        { "ref": [ "descr" ] },
        { "ref": [ "supplier_ID" ] }
      ] }
    ]
  }
```

↳ *See an example how to handle expands in Node.js.*

Expands across local and remote can cause stability and performance issues. For a list of items, you need to collect all IDs and send it to the database or the remote system. This can become long and may exceed the limits of a URL string in case of OData. Do you really need expands for a list of items?

```http
GET /service/risk/Risks?$expand=supplier
```

Or is it sufficient for single items?

```http
GET /service/risk/Risks(545A3CF9-84CF-46C8-93DC-E29F0F2BC6BE)/?$expand=supplier
```

> **Keep performance in mind**
>
> Consider to reject expands if it's requested on a list of items.

Handle Navigations Across Local and Remote Entities

Navigations allow to address items via an association from a different entity:

```http
GET /service/risks/Risks(20466922-7d57-4e76-b14c-e53fd97dcb11)/supplier
```

The CQN consists of a `from` condition with 2 values for `ref`. The first `ref` selects the record of the source entity of the navigation. The second `ref` selects the name of the association, to navigate to the target entity.

```json
{
  "from": {
    "ref": [ {
      "id": "RiskService.Risks",
```

```
        "where": [
          { "ref": [ "ID" ] },
          "=",
          { "val": "20466922-7d57-4e76-b14c-e53fd97dcb11" }
        ]},
        "supplier"
      ]
    },
    "columns": [
      { "ref": [ "ID" ] },
      { "ref": [ "fullName" ] },
      { "ref": [ "isBlocked" ] }
    ],
    "one": true
  }
```

To handle navigations, you need to check in your code if the `from.ref` object contains 2 elements. Be aware, that for navigations the handler of the **target** entity is called.

If the association's on condition equals the key of the source entity, you can directly select the target entity using the key's value. You find the value in the `where` block of the first `from.ref` entry.

Otherwise, you need to select the source item using that `where` block and take the required fields for the associations on condition from that result.

↳ *See an example how to handle navigations in Node.js.*

Limitations and Feature Matrix

Required Implementations for Mashups

You need additional logic, if remote entities are in the game. The following table shows what is required. "Local" is a database entity or a projection on a database entity.

Request	Example	Implementation
Local (including navigations and expands)	`/service/risks/Risks`	Handled by CAP

Request	Example	Implementation
Local: Expand remote	`/service/risks/Risks?$expand=supplier`	Delegate query w/o expand to local service and implement expand.
Local: Navigate to remote	`/service/risks(...)/supplier`	Implement navigation and delegate query target to remote service.
Remote (including navigations and expands to the same remote service)	`/service/risks/Suppliers`	Delegate query to remote service
Remote: Expand local	`/service/risks/Suppliers?$expand=risks`	Delegate query w/o expand to remote service and implement expand.
Remote: Navigate to local	`/service/Suppliers(...)/risks`	Implement navigation, delegate query for target to local service

Transient Access vs. Replication

> This chapter shows only techniques for transient access.

The following matrix can help you to find the best approach for your scenario:

Feature	Transient Access	Replication
Filtering on local **or** remote fields [1]	Possible	Possible
Filtering on local **and** remote fields [2]	Not possible	Possible
Relationship: Uni-/Bidirectional associations	Possible	Possible
Relationship: Flatten	Not possible	Possible
Evaluate user permissions in remote system	Possible	Requires workarounds [3]
Data freshness	Live data	Outdated until replicated
Performance	Degraded [4]	Best

> [1] It's **not required** to filter both, on local and remote fields, in the same request.
> [2] It's **required** to filter both, on local and remote fields, in the same request.
> [3] Because replicated data is accessed, the user permission checks of the remote

> system aren't evaluated.
> [4] Depends on the connectivity and performance of the remote system.

Connect and Deploy

Using Destinations

Destinations contain the necessary information to connect to a remote system. They're basically an advanced URL, that can carry additional metadata like, for example, the authentication information.

You can choose to use SAP BTP destinations or application defined destinations.

Use SAP BTP Destinations

CAP leverages the destination capabilities of the SAP Cloud SDK.

Create Destinations on SAP BTP

Create a destination using one or more of the following options.

- **Register a system in your global account:** You can check here how to Register an SAP System in your SAP BTP global account and which systems are supported for registration. Once the system is registered and assigned to your subaccount, you can create a service instance. A destination is automatically created along with the service instance.
- **Connect to an on-premise system:** With SAP BTP Cloud Connector , you can create a connection from your cloud application to an on-premise system.
- **Manually create destinations:** You can create destinations manually in your SAP BTP subaccount. See section destinations in the SAP BTP documentation.
- **Create a destination to your application:** If you need a destination to your application, for example, to call it from a different application, then you can automatically create it in the MTA deployment.

Use Destinations with Node.js

In your *package.json*, a configuration for the `API_BUSINESS_PARTNER` looks like this:

```json
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      "kind": "odata",
      "model": "srv/external/API_BUSINESS_PARTNER"
    }
  }
}
```

If you've imported the external service definition using `cds import`, an entry for the service in the *package.json* has been created already. Here you specify the name of the destination in the `credentials` block.

In many cases, you also need to specify the `path` prefix to the service, which is added to the destination's URL. For services listed on the SAP Business Accelerator Hub, you can find the path in the linked service documentation.

Since you don't want to use the destination for local testing, but only for production, you can profile it by wrapping it into a `[production]` block:

```json
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      "kind": "odata",
      "model": "srv/external/API_BUSINESS_PARTNER",
      "[production]": {
        "credentials": {
          "destination": "S4HANA",
          "path": "/sap/opu/odata/sap/API_BUSINESS_PARTNER"
        }
      }
    }
  }
}
```

Additionally, you can provide destination options inside a `destinationOptions` object:

```jsonc
"cds": {
  "requires": {
    "API_BUSINESS_PARTNER": {
      /* ... */
      "[production]": {
```

```
        "credentials": {
          /* ... */
        },
        "destinationOptions": {
          "selectionStrategy": "alwaysSubscriber",
          "useCache": true
        }
      }
    }
  }
}
```

The `selectionStrategy` property controls how a destination is resolved.

The `useCache` option controls whether the SAP Cloud SDK caches the destination. It's enabled by default but can be disabled by explicitly setting it to `false`. Read Destination Cache to learn more about how the cache works.

If you want to configure additional headers for the HTTP request to the system behind the destination, for example an Application Interface Register (AIR) header, you can specify such headers in the destination definition itself using the property *URL.headers.<header-key>*.

Use Application Defined Destinations

If you don't want to use SAP BTP destinations, you can also define destinations, which means the URL, authentication type, and additional configuration properties, in your application configuration or code.

Application defined destinations support a subset of properties and authentication types of the SAP BTP destination service.

Configure Application Defined Destinations in Node.js

You specify the destination properties in `credentials` instead of putting the name of a destination there.

This is an example of a destination using basic authentication:

jsonc

```
"cds": {
  "requires": {
    "REVIEWS": {
      "kind": "odata",
      "model": "srv/external/REVIEWS",
```

```
      "[production]": {
        "credentials": {
          "url": "https://reviews.ondemand.com/reviews",
          "authentication": "BasicAuthentication",
          "username": "<set from code or env>",
          "password": "<set from code or env>",
          "headers": {
            "my-header": "header value"
          },
          "queries": {
            "my-url-param": "url param value"
          }
        }
      }
    }
  }
}
```

↳ *Supported destination properties.*

> **WARNING**
>
> You shouldn't put any sensitive information here.

Instead, set the properties in the bootstrap code of your CAP application:

```js
const cds = require("@sap/cds");

if (cds.env.requires?.credentials?.authentication === "BasicAuthentication") {
  const credentials = /* read your credentials */
  cds.env.requires.credentials.username = credentials.username;
  cds.env.requires.credentials.password = credentials.password;
}
```

You might also want to set some values in the application deployment. This can be done using env variables. For this example, the env variable for the URL would be `cds_requires_REVIEWS_credentials_destination_url`.

This variable can be parameterized in the *manifest.yml* for a `cf push` based deployment:

manifest.yml

```yaml
applications:
- name: reviews
  ...
  env:
    cds_requires_REVIEWS_credentials_url: ((reviews_url))
```

```sh
cf push --var reviews_url=https://reviews.ondemand.com/reviews
```

The same can be done using *mtaext* file for MTA deployment.

If the URL of the target service is also part of the MTA deployment, you can automatically receive it as shown in this example:

mta.yaml

```yaml
- name: reviews
  provides:
   - name: reviews-api
     properties:
       reviews-url: ${default-url}
- name: bookshop
  requires:
   ...
   - name: reviews-api
  properties:
    cds_requires_REVIEWS_credentials_url: ~{reviews-api/reviews-url}
```

.env

```properties
cds_requires_REVIEWS_credentials_url=http://localhost:4008/reviews
```

> **WARNING**
>
> For the *configuration path*, you **must** use the underscore (" _ ") character as delimiter. CAP supports dot (" . ") as well, but Cloud Foundry won't recognize variables using dots. Your *service name* **mustn't** contain underscores.

Implement Application Defined Destinations in Node.js

There is no API to create a destination in Node.js programmatically. However, you can change the properties of a remote service before connecting to it, as shown in the

previous example.

Connect to Remote Services Locally

If you use SAP BTP destinations, you can access them locally using CAP's hybrid testing capabilities with the following procedure:

Bind to Remote Destinations

Your local application needs access to an XSUAA and Destination service instance in the same subaccount where the destination is:

1. Login to your Cloud Foundry org and space
2. Create an XSUAA service instance and service key:

```sh
cf create-service xsuaa application cpapp-xsuaa
cf create-service-key cpapp-xsuaa cpapp-xsuaa-key
```

3. Create a Destination service instance and service key:

```sh
cf create-service destination lite cpapp-destination
cf create-service-key cpapp-destination cpapp-destination-key
```

4. Bind to XSUAA and Destination service:

```sh
cds bind -2 cpapp-xsuaa,cpapp-destination
```

↳ *Learn more about `cds bind`.*

Run a Node.js Application with a Destination

Add the destination for the remote service to the `hybrid` profile in the *.cdsrc-private.json* file:

```jsonc
{
  "requires": {
    "[hybrid]": {
      "auth": {
        /* ... */
      },
```

```
        "destinations": {
          /* ... */
        },
        "API_BUSINESS_PARTNER": {
          "credentials": {
            "path": "/sap/opu/odata/sap/API_BUSINESS_PARTNER",
            "destination": "cpapp-bupa"
          }
        }
      }
    }
  }
```

Run your application with the Destination service:

```sh
cds watch --profile hybrid
```

> **TIP**
>
> If you are developing in the Business Application Studio and want to connect to an on-premise system, you will need to do so via Business Application Studio's built-in proxy, for which you need to add configuration in an `.env` file. See Connecting to External Systems From the Business Application Studio for more details.

Connect to an Application Using the Same XSUAA (Forward Authorization Token)

If your application consists of microservices and you use one (or more) as a remote service as described in this guide, you can leverage the same XSUAA instance. In that case, you don't need an SAP BTP destination at all.

Assuming that your microservices use the same XSUAA instance, you can just forward the authorization token. The URL of the remote service can be injected into the application in the MTA or Cloud Foundry deployment using application defined destinations.

Forward Authorization Token with Node.js

To enable the token forwarding, set the `forwardAuthToken` option to `true` in your application defined destination:

```json
{
  "requires": {
    "kind": "odata",
    "model": "./srv/external/OrdersService",
    "credentials": {
      "url": "<set via env var in deployment>",
      "forwardAuthToken": true
    }
  }
}
```

Connect to an Application in Your Kyma Cluster

The Istio service mesh provides secure communication between the services in your service mesh. You can access a service in your applications' namespace by just reaching out to `http://<service-name>` or using the full hostname `http://<service-name>.<namespace>.svc.cluster.local`. Istio sends the requests through an mTLS tunnel.

With Istio, you can further secure the communication by configuring authentication and authorization for your services

Deployment

Your micro service needs bindings to the **XSUAA** and **Destination** service to access destinations on SAP BTP. If you want to access an on-premise service using **Cloud Connector**, then you need a binding to the **Connectivity** service as well.

↳ *Learn more about deploying CAP applications.*

↳ *Learn more about deploying an application using the end-to-end tutorial.*

Add Required Services to MTA Deployments

The MTA-based deployment is described in the deployment guide. You can follow this guide and make some additional adjustments to the generated *mta.yml* file.

```sh
cds add xsuaa,destination,connectivity
```

▶ *Learn what this does in the background...*

Build your application:

```sh
mbt build -t gen --mtar mta.tar
```

Now you can deploy it to Cloud Foundry:

```sh
cf deploy gen/mta.tar
```

Connectivity Service Credentials on Kyma

The secret of the connectivity service on Kyma needs to be modified for the Cloud SDK to connect to on-premise destinations.

↳ *Support for Connectivity Service Secret in Node.js*

Destinations and Multitenancy

With the destination service, you can access destinations in your provider account, the account your application is running in, and destinations in the subscriber accounts of your multitenant-aware application.

Use Destinations from Subscriber Account

Customers want to see business partners from, for example, their SAP S/4 HANA system.

As provider, you need to define a name for a destination, which enables access to systems of the subscriber of your application. In addition, your multitenant application or service needs to have a dependency to the destination service. For destinations in an on-premise system, the connectivity service must be bound.

The subscriber needs to create a destination with that name in their subscriber account, for example, pointing to their SAP S/4HANA system.

Destination Resolution

The destination is read from the tenant of the request's JWT (authorization) token. If no JWT token is present *or the destination isn't found*, the destination is read from the tenant of the application's XSUAA binding.

> **JWT token vs. XSUAA binding**

> Using the tenant of the request's JWT token means reading from the **subscriber subaccount** for a multitenant application. The tenant of the application's XSUAA binding points to the destination of the **provider subaccount**, the account where the application is deployed to.

You can change the destination lookup behavior as follows:

```jsonc
"cds": {
  "requires": {
    "SERVICE_FOR_PROVIDER": {
      /* ... */
      "credentials": {
        /* ... */
      },
      "destinationOptions": {
        "selectionStrategy": "alwaysProvider",
        "jwt": null
      }
    }
  }
}
```

Setting the `selectionStrategy` property for the destination options to `alwaysProvider`, you can ensure that the destination is always read from your provider subaccount. With that you ensure that a subscriber cannot overwrite your destination.

Set the destination option `jwt` to `null`, if you don't want to pass the request's JWT to SAP Cloud SDK. Passing the request's JWT to SAP Cloud SDK has implications on, amongst others, the effective defaults for selection strategy and isolation level. In rare cases, these defaults are not suitable, for example when the request to the upstream server does not depend on the current user. Please see Authentication and JSON Web Token (JWT) Retrieval for more details.

Add Qualities

Resilience

There are two ways to make your outbound communications resilient:

1. Run your application in a service mesh (for example, Istio, Linkerd, etc.). For example, Kyma is provided as service mesh.
2. Implement resilience in your application.

Refer to the documentation for the service mesh of your choice for instructions. No code changes should be required.

To build resilience into your application, there are libraries to help you implement functions, like doing retries, circuit breakers or implementing fallbacks.

There's no resilience library provided out of the box for CAP Node.js. However, you can use packages provided by the Node.js community. Usually, they provide a function to wrap your code that adds the resilience logic.

Resilience in Kyma

Kyma clusters run an Istio service mesh. Istio allows to configure resilience for the network destinations of your service mesh.

Tracing

CAP adds headers for request correlation to its outbound requests that allows logging and tracing across micro services.

↳ *Learn more about request correlation in Node.js.*

Feature Details

Legend

Tag	Explanation
✓	supported
✗	not supported

Supported Protocols

Protocol	Java	Node.js
odata-v2	✓	✓
odata-v4	✓	✓
rest	✗	✓

> **TIP**
>
> The Node.js runtime supports `odata` as an alias for `odata-v4` as well.

Querying API Features

Feature	Java	Node.js
READ	✓	✓
INSERT/UPDATE/DELETE	✓	✓
Actions	✓	✓
`columns`	✓	✓
`where`	✓	✓
`orderby`	✓	✓
`limit` (top & skip)	✓	✓
`$apply` (aggregate, groupby, ...)	✗	✗
`$search` (OData v4)	✓	✓
`search` (SAP OData v2 extension)	✓	✓

Supported Projection Features

Feature	Java	Node.js
Resolve projections to remote services	✓	✓
Resolve multiple levels of projections to remote services	✓	✓

Feature	Java	Node.js
Aliases for fields	✓	✓
`excluding`	✓	✓
Resolve associations (within the same remote service)	✓	✓
Redirected associations	✓	✓
Flatten associations	✗	✗
`where` conditions	✗	✗
`order by`	✗	✗
Infix filter for associations	✗	✗
Model Associations with mixins	✓	✓

Supported Features for Application Defined Destinations

The following properties and authentication types are supported for *application defined destinations*:

Properties

These destination properties are fully supported by both, the Java and the Node.js runtime.

> **TIP**
>
> This list specifies the properties for application defined destinations.

Properties	Description
`url`	
`authentication`	Authentication type
`username`	User name for BasicAuthentication
`password`	Password for BasicAuthentication
`headers`	Map of HTTP headers
`queries`	Map of URL parameters

Properties	Description
`forwardAuthToken`	Forward auth token

↳ *Destination Type in SAP Cloud SDK for JavaScript*

Authentication Types

Authentication Types	Java	Node.js
NoAuthentication	✓	✓
BasicAuthentication	✓	✓
TokenForwarding	✓	✗ Use `forwardAuthToken`
OAuth2ClientCredentials	code only	✗
UserTokenAuthentication	code only	✗