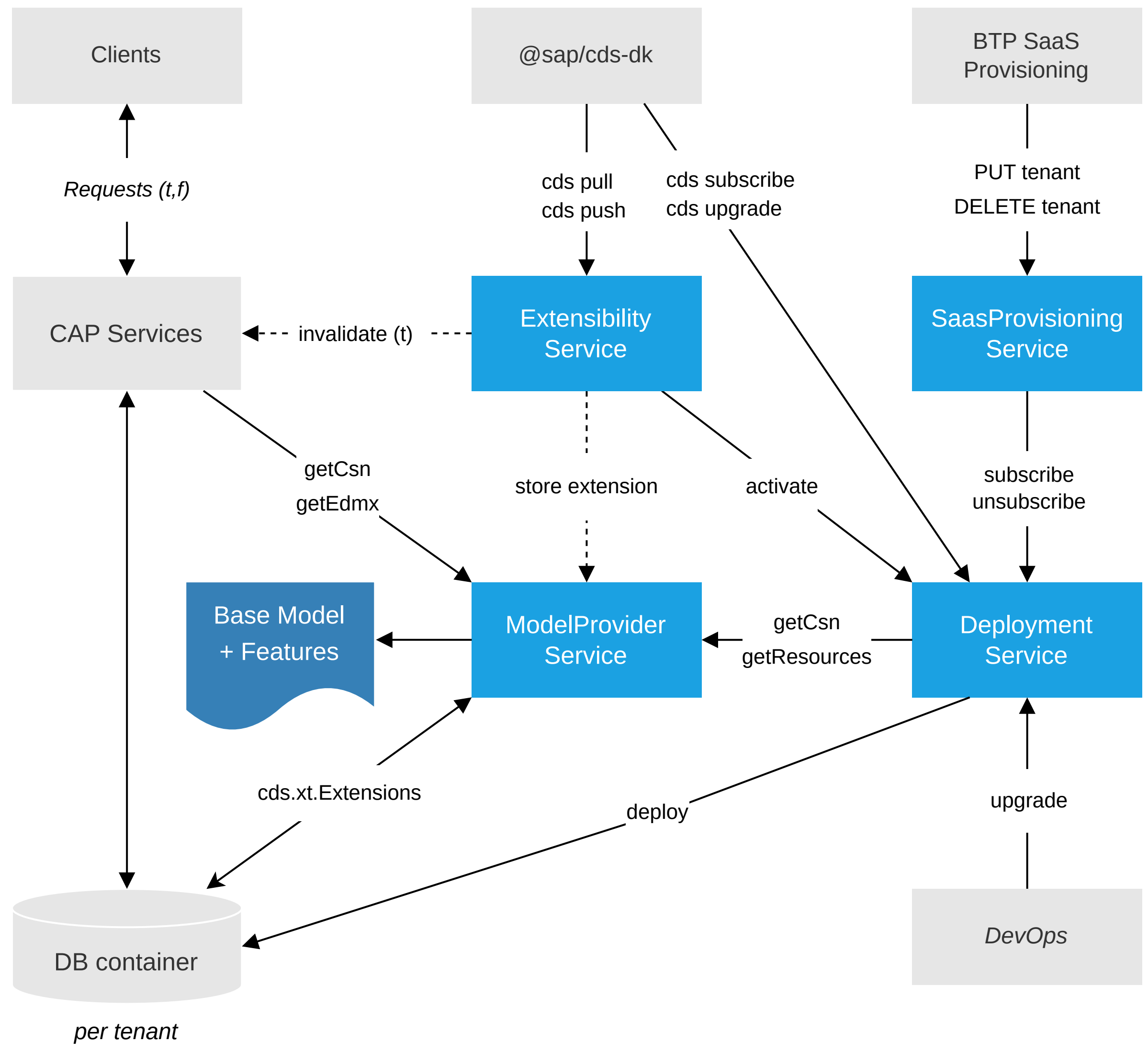
MTX Services Reference

Introduction & Overview

The `@sap/cds-mtxs` package provides a set of CAP services which implement ***multitenancy***, *features toggles* and *extensibility* (*'MTX'* stands for these three functionalities). These services work in concert as depicted in the following diagram:



MTX services are implemented in Node.js and can run in the same Node.js server as your application services or in separate micro services called *sidecars*. All services can be consumed via REST APIs.

As the services are defined and implemented as standard CAP services, with definitions in CDS and implementations based on the CAP Node.js framework, application projects can hook into all events to add custom logic using CAP Node.js.

Getting Started...

Add `@sap/cds-mtxs` Package Dependency

sh

```sh
npm add @sap/cds-mtxs
```

Enable MTX Functionality

Add one or more of the following convenience configuration flags, for example, to your *package.json* in a Node.js-based project:

```json
"cds": {
  "requires": {
    "multitenancy": true,
    "extensibility": true,
    "toggles": true
  }
}
```

↳ *Java-based projects require a sidecar setup.*

Test-Drive Locally

After enabling MTX features, you can test MTX functionality with local development setups and in-memory databases as usual:

```sh
cds watch
```

This shows the MTX services being served in addition to your app services:

```log
[cds] - loaded model from 6 file(s):

  db/schema.cds
  srv/admin-service.cds
  srv/cat-service.cds
  ../../db/extensions.cds
  ../../srv/deployment-service.cds
  ../../srv/bootstrap.cds

[cds] - connect to db > sqlite { url: ':memory:' }
[cds] - serving cds.xt.SaasProvisioningService { path: '/-/cds/saas-provision
[cds] - serving cds.xt.DeploymentService { path: '/-/cds/deployment' }
[cds] - serving cds.xt.ModelProviderService { path: '/-/cds/model-provider' }
[cds] - serving cds.xt.ExtensibilityService { path: '/-/cds/extensibility' }
[cds] - serving cds.xt.JobsService { path: '/-/cds/jobs' }
[cds] - serving AdminService { path: '/admin' }
[cds] - serving CatalogService { path: '/browse', impl: 'srv/cat-service.js'
```

```
[cds] - server listening on { url: 'http://localhost:4004' }
[cds] - launched at 5/6/2023, 9:31:11 AM, in: 863.803ms
```

Grow As You Go

Follow CAP principles of *'Grow as you go...'* to minimize complexity of setups, stay in inner loops with fast turnarounds, and hence minimize costs and accelerate development.

Enable MTX Only if Required

During development you rarely need to run your servers with MTX functionality enabled. Only do so when you really need it. For example, in certain tests or by using configuration profiles.

This configuration would have development not use MTX by default. You could still run with MTX enabled on demand and have it always active in production:

```jsonc
"cds": {
  "requires": {
    "[local-multitenancy]": {
       "multitenancy": true,
       "extensibility": true,
       "toggles": true
    },
    "[production]": {
       "multitenancy": true,
       "extensibility": true,
       "toggles": true
    }
  }
}
```

During development you could occasionally run with MTX:

```sh
cds watch --profile local-multitenancy
```

Testing With Minimal Setup

When designing test suites that run frequently in CI/CD pipelines, you can shorten runtimes and reduce costs. First run a set of functional tests which use MTX in minimized setups – that is, with local servers and in-memory databases as introduced in the *Multitenancy* guide.

Only in the second and third phases, you would then run the more advanced hybrid tests. These hybrid tests could include testing tenant subscriptions with SAP HANA, or integration tests with the full set of required cloud services.

Sidecar Setups

In the minimal setup introduced in the *Getting Started...* chapter, we had the MTX services being served embedded with our main app, that is, in the same server as our application services. While this is possible for Node.js and even recommended to reduce complexity during development, quite frequently, we'd want to run them in a separate micro service. Reasons for that include:

- **For Java-based projects** — As these services are implemented in Node.js we need to run them separately and consume them remotely for Java-based apps.
- **To scale independently** — As some operations, especially `upgrade`, are very resource-intensive, we want to scale these services separate from our main application.

As MTX services are built and consumed as CAP services, we benefit from CAP's agnostic design and can easily move them to separate services.

Create Sidecar as a Node.js Subproject

An MTX sidecar is a standard, yet minimal Node.js CAP project. By default it's added to a subfolder `mtx/sidecar` within your main project, containing just a *package.json* file.

mtx/sidecar/package.json

```json
{
  "name": "bookshop-mtx", "version": "0.0.0",
  "dependencies": {
    "@sap/cds": "^9",
    "@cap-js/hana": "^2",
    "@sap/cds-mtxs": "^3",
    "@sap/xssec": "^4",
    "express": "^4"
  },
  "devDependencies": {
    "@cap-js/sqlite": "^2"
  },
  "scripts": {
    "start": "cds-serve"
  },
  "cds": {
    "profile": "mtx-sidecar"
  }
}
```

The only configuration necessary for the project is the *`mtx-sidecar`* profile.

▶ *Let's have a look at what this profile provides...*

Testing Sidecar Setups

With the above setup in place, we can test-drive the sidecar mode locally. To do so, we'll simply start the sidecar and main app in separate shells.

1 Run sidecar in first shell:

```sh
cds watch mtx/sidecar
```

▶ *You see the sidecar starting on port 4005...*

2 Run the main app as before in a second shell:

```sh
cds watch
```

ModelProviderService serving models from main app

When we use our application, we can see `model-provider/getCsn` requests in the sidecar's trace log. In response to those requests, the sidecar reads and returns the main app's models, that is, the models from two levels up the folder hierarchy as is the default with the `mtx-sidecar` profile.

Note: Service Bindings by `cds watch`

Required service bindings are done automatically by `cds watch`'s built-in runtime service registry. This is how it works:

1. Each server started using `cds watch` registers all served services in `~/cds-services.json`.
2. Every subsequently started server binds automatically all `required` remote services, to equally named services already registered in `~/cds-services.json`.

In our case: The main app's `ModelProviderService` automatically receives the service binding credentials, for example `url`, to talk to the one served by the sidecar.

Build Sidecar for Production

When deploying a sidecar for production, it doesn't have access to the main app's models two levels up the deployed folder hierarchy. Instead we have to prepare deployment by running `cds build` in the project's root:

```sh
cds build
```

One of the build tasks that are executed is the `mtx-sidecar` build task. It generates log output similar to the following:

```log
[cds] - the following build tasks will be executed
  {"for":"mtx-sidecar", "src":"mtx/sidecar", "options":... }
[cds] - done > wrote output to:
   gen/mtx/sidecar/_main/fts/isbn/csn.json
   gen/mtx/sidecar/_main/fts/reviews/csn.json
   gen/mtx/sidecar/_main/resources.tgz
```

```
    gen/mtx/sidecar/_main/srv/_i18n/i18n.json
    gen/mtx/sidecar/_main/srv/csn.json
    gen/mtx/sidecar/package.json
    gen/mtx/sidecar/srv/_i18n/i18n.json
    gen/mtx/sidecar/srv/csn.json
[cds] - build completed in 687 ms
```

The outcome of that build task is a compiled and deployable version of the sidecar in the *gen/mtx/sidecar* staging areas:

```zsh
bookshop/
├─ _i18n/
├─ app/
├─ db/
├─ fts/
├─ gen/mtx/sidecar/
│   ├─ _main/
│   │   ├── fts/
│   │   │   ├── isbn/
│   │   │   │   └── csn.json
│   │   │   └── reviews/
│   │   │       └── csn.json
│   │   ├── srv/
│   │   │   ├── _i18n
│   │   │   └── csn.json
│   │   └── resources.tgz
│   └─ package.json
├─ mtx/sidecar/
├─ ...
```

In essence, the `mtx-sidecar` build task does the following:

1. It runs a standard Node.js build for the sidecar.
2. It pre-compiles the main app's models, including all features into respective *csn.json* files, packaged into the `_main` subfolder.
3. It collects all additional sources required for subsequent deployments to `resources.tgz`. For example, these include *.csv* and *i18n* files.

Test-Drive Production Locally

We can also test-drive the production-ready variant of the sidecar locally before actual deployment, again using two separate shells.

1. **First, start sidecar** from `gen/mtx/sidecar` in `prod` simulation mode:

   sh
   ```sh
   cds watch gen/mtx/sidecar --profile development,prod
   ```

2. **Second, start main** app as usual:

   sh
   ```sh
   cds watch
   ```

ModelProviderService serving models from main app

When we now use our application again, and inspect the sidecar's trace logs, we see that the sidecar reads and returns the main app's precompiled models from `_main` now:

log
```log
[cds] – POST /-/cds/model-provider/getCsn
[cds] – model loaded from 3 file(s):

  gen/mtx/sidecar/_main/srv/csn.json
  gen/mtx/sidecar/_main/fts/isbn/csn.json
  gen/mtx/sidecar/_main/fts/reviews/csn.json
```

Configuration

Shortcuts `cds.requires.multitenancy / extensibility / toggles`

The easiest way to enable multitenancy, extensibility, and feature toggles is as follows:

json
```json
"cds": {
  "requires": {
    "multitenancy": true,
    "extensibility": true,
    "toggles": true
  }
}
```

On the one hand, these settings are interpreted by the CAP runtime to support features such as tenant-specific database connection pooling when `multitenancy` is enabled.

On the other hand, these flags are checked during server bootstrapping to ensure the required combinations of services are served by default. The following tables shows which services are enabled by one of the shortcuts:

	`multitenancy`	`extensibility`	`toggles`
SaasProvisioningService	yes	no	no
DeploymentService	yes	no	no
ExtensibilityService	no	yes	no
ModelProviderService	yes	yes	yes

Configuring Individual Services

In addition or alternatively to the convenient shortcuts above you can configure each service individually, as shown in the following examples:

```jsonc
"cds": {
  "requires": {
    "cds.xt.DeploymentService": true
  }
}
```

The names of the service-individual configuration options are:

- `cds/requires/<service definition name>`

Allowed Values

- `false` — deactivates the service selectively
- `true` — activates the service with defaults for embedded usage
- `<preset name>` — uses preset, for example, with defaults for sidecar usage
- `{ ...options }` — add/override individual configuration options

Common Config Options

- `model` — specifies/overrides the service model to be used

- `impl` — specifies/overrides the service implementation to be used
- `kind` — the kind of service/consumption, for example, `rest` for remote usage

> These options are supported by all services.

Combined with Convenience Flags

json
```json
"cds": {
  "requires": {
    "multitenancy": true,
    "cds.xt.SaasProvisioningService": false,
    "cds.xt.DeploymentService": false,
    "cds.xt.ModelProviderService": { "kind": "rest" }
  }
}
```

This tells the CAP runtime to enable multitenancy, but neither serve the *DeploymentService*, nor the *SaasProvisioningService*, and to use a remote *ModelProviderService* via REST protocol.

Individual Configurations Only

We can also use only the individual service configurations:

json
```json
"cds": {
  "requires": {
    "cds.xt.DeploymentService": true,
    "cds.xt.ModelProviderService": { "root": "../.." }
  }
}
```

In this case, the server will **not** run in multitenancy mode. Also, extensibility and feature toggles are not supported. Yet, the *DeploymentService* and the *ModelProviderService* are served selectively. For example, this kind of configuration can be used in sidecars.

Using Configuration Presets

Profile-based configuration

The simplest and for most projects sufficient configuration is the profile-based one, where just these two entries are necessary:

package.json

```json
"cds": {
  "profile": "with-mtx-sidecar"
}
```

mtx/sidecar/package.json

```json
"cds": {
  "profile": "mtx-sidecar"
}
```

Preset-based configuration

Some MTX services come with pre-defined configuration presets, which can easily be used by referring to the preset suffixes. For example, to simplify and standardize sidecar configuration, *ModelProviderService* supports the `in-sidecar` preset which can be used like that:

```json
  "cds": {
    "requires": {
      "cds.xt.ModelProviderService": "in-sidecar"
    }
  }
```

These presets are actually configured in `cds.env` defaults like that:

```js
cds: {
  requires: {
    // Configuration Presets (in cds.env.requires.kinds)
    kinds: {
      "cds.xt.ModelProviderService-in-sidecar": {
        "[development]": { root: "../.." },
        "[production]": { root: "_main" },
      },
      "cds.xt.ModelProviderService": {
        model: "@sap/cds/srv/model-provider"
      },
      // ...
    }
  }
}
```

↳ *Learn more about `cds.env`*

Inspecting Effective Configuration

You can always inspect the effective configuration by executing this in the *mtx/sidecar* folder:

```sh
cds env get requires
```

This will give you an output like this:

```js
{
  auth: { strategy: 'dummy', kind: 'dummy' },
  'cds.xt.ModelProviderService': {
    root:'../..',
    model:'@sap/cds/srv/model-provider',
    kind:'in-sidecar'
  }
}
```

Add CLI option `--profile` to inspect configuration in different profiles:

```sh
cds env get requires --profile development
cds env get requires --profile production
```

Customization

All services are defined and implemented as standard CAP services, with service definitions in CDS, and implementations based on the CAP Node.js framework. Thus, you can easily do both, adapt service definitions, as well as hook into all events to add custom logic using CAP Node.js.

Customizing Service Definitions

For example, you could override the endpoints to serve a service:

```cds
using { cds.xt.ModelProviderService } from '@sap/cds-mtxs';
annotate ModelProviderService with @path: '/mtx/mps';
```

For sidecar scenarios, define the annotations in the Node.js sidecar application and not as part of the main application.

Adding Custom Lifecycle Event Handlers

Register handlers in *server.js* files:

mtx/sidecar/server.js

```js
const cds = require('@sap/cds')
cds.on('served', ()=>{
  const { 'cds.xt.ModelProviderService': mps } = cds.services
  const { 'cds.xt.DeploymentService': ds } = cds.services
  ds.before ('upgrade', (req) => { ... })
  ds.after ('subscribe', (_,req) => { ... })
  mps.after ('getCsn', (csn) => { ... })
})
```

> **Custom hooks for CLI usage**
>
> For CLI usage via *cds subscribe|upgrade|unsubscribe* you can create a *mtx/sidecar/cli.js* file, which works analogously to a *server.js* .

Consumption

Via Programmatic APIs

Consume MTX services using standard Service APIs. For example, in `cds repl`:

```js
await cds.test()
var { 'cds.xt.ModelProviderService': mps } = cds.services
var { 'cds.xt.DeploymentService': ds } = cds.services
var db = await ds.subscribe ('t1')
var csn = await mps.getCsn('t1')
cds.context = { tenant:'t1' }
await db.run('SELECT type, name from sqlite_master')
```

Via REST APIs

Common usage of the MTX services is through REST APIs. Here's an example:

1. Start the server

   ```sh
   cds watch
   ```

2. Subscribe a tenant

   ```http
   POST /-/cds/deployment/subscribe HTTP/1.1
   Content-Type: application/json

   {
     "tenant": "t1"
   }
   ```

3. Get CSN from `ModelProviderService`

   ```http
   POST /-/cds/model-provider/getCsn HTTP/1.1
   Content-Type: application/json

   {
     "tenant": "t1",
     "toggles": ["*"]
   }
   ```

ModelProviderService

The *ModelProviderService* serves model variants, which may include tenant-specific extensions and/or feature-toggled aspects.

Service Definition	`@sap/cds-mtxs/srv/model-provider`
Service Definition Name	`cds.xt.ModelProviderService`
Default HTTP Endpoint	`/-/cds/model-provider`

Configuration

```json
"cds.xt.ModelProviderService": {
  "root": "../../custom/path"
}
```

- Common Config Options
- `root` — a directory name, absolute or relative to the *package.json*'s location, specifying the location to search for models and resources to be served by the model provider services. Default is undefined, for embedded usage of model provider. In case of a sidecar, it refers to the main app's model; usually `"../.."` during development, and `"_main"` in production.

Supported Presets

- `in-sidecar` — provides defaults for usage in sidecars
- `from-sidecar` — shortcut for `{ "kind": "rest" }`

`getCsn` *(tenant, toggles) → CSN*

Returns the application's effective CSN document for the given tenant + feature toggles vector. CAP runtimes call that method to obtain the effective models to serve.

Arguments	Description
`tenant`	A string identifying the tenant

Arguments	Description
toggles	An array listing toggled features; `['*']` for all features

Example Usage

```http
POST /-/cds/model-provider/getCsn HTTP/1.1
Content-Type: application/json

{
  "tenant": "t1",
  "toggles": ["*"]
}
```

The response is a CSN in JSON representation.

↳ *Learn more about **CSN***

`getEdmx` *(tenant, toggles, service, locale) → EDMX*

Returns the EDMX document for a given service in context of the given tenant and feature toggles vector. CAP runtimes call this to get the EDMX document they return in response to OData `$metadata` requests.

Arguments	Description
tenant	A string identifying the tenant
toggles	An array listing toggled features; `['*']` for all features
service	Fully-qualified name of a service definition
locale	Requested locale, that is, as from `accept-language` header

Example Usage

```http
POST /-/cds/model-provider/getEdmx HTTP/1.1
Content-Type: application/json

{
  "tenant": "t1",
  "toggles": ["*"],
  "service": "CatalogService",
  "locale": "en"
}
```

`getResources` *() → TAR*

Returns a *.tar* archive containing CSV files, I18n files, as well as native database artifacts, required for deployment to databases. `DeploymentService` calls that whenever it receives a `subscribe` or `upgrade` event.

`getExtensions` *(tenant) → CSN*

Returns a *parsed* CSN document containing all the extensions stored in `cds.xt.Extensions` for the given tenant.

Arguments	Description
`tenant`	A string identifying the tenant

`isExtended` *(tenant) → true|false*

Returns `true` if the given `tenant` has extensions applied.

Arguments	Description
`tenant`	A string identifying the tenant

ExtensibilityService

The *ExtensibilityService* allows to add and activate tenant-specific extensions at runtime.

Service Definition	`@sap/cds-mtxs/srv/extensibility-service`

Service Definition Name	*cds.xt.ExtensibilityService*
Default HTTP Endpoint	*/-/cds/extensibility*

↳ *See the extensibility guide for more context*

Configuration

jsonc

```jsonc
"cds.xt.ExtensibilityService": {
  // fields must start with x_ or xx_
  "element-prefix": ["x_", "xx_"],
   // namespaces starting with com.sap or sap. can't be extended
  "namespace-blocklist": ["com.sap.", "sap."],
  "extension-allowlist": [
    {
      // at most 2 new fields in entities from the my.bookshop namespace
      "for": ["my.bookshop"],
      "kind": "entity",
      "new-fields": 2,
      // allow extensions for field "description" only
      "fields": ["description"]
    },
    {
      // at most 2 new entities in CatalogService
      "for": ["CatalogService"],
      "new-entities": 2,
      // allow @readonly annotations in CatalogService
      "annotations": ["@readonly"]
    }
  ]
}
```

- Common Config Options
- *element-prefix* — restrict field names to prefix
- *namespace-blocklist* — restrict namespaces to be extended
- *extension-allowlist* — allow certain entities to be extended

> Without *extension-allowlist* configured, extensions are forbidden.

Using *"for": ["*"]* applies the rules to all possible values.

↳ *See the list of possible `kind` values.*

- `new-fields` specifies the maximum number of fields that can be added.
- `fields` lists the fields that are allowed to be extended. If the list is omitted, all fields can be extended.
- `new-entities` specifies the maximum number of entities that can be added to a service.

↳ *Check Extension Restrictions for more details.*

GET `Extensions/<ID>` → *[{ ID, csn, timestamp }]*

Returns a list of all tenant-specific extensions.

Request Format

Parameters	Description
`ID`	String uniquely identifying the extension

> Omitting `ID` will return all extensions.

Response Format

Body	Description
`ID`	String uniquely identifying the extension
`csn`	Compiled extension CSN
`timestamp`	Timestamp of activation date

Example Request

Get a specific extension

Request Response

```http
GET /-/cds/extensibility/Extensions/isbn-extension HTTP/1.1
Content-Type: application/json
```

Get all extensions

Request Response

```http
GET /-/cds/extensibility/Extensions HTTP/1.1
```

```
Content-Type: application/json
```

PUT `Extensions/<ID>` ([csn]) → *[{ ID, csn, timestamp }]*

Creates a new tenant-specific extension.

HTTP Request Options

Request Header	Example Value	Description
prefer	*respond-async*	Trigger asynchronous extension activation

Request Format

Parameters	Description
ID	String uniquely identifying the extension
Body	
csn	Array of extension CDL or CSN to apply
i18n	Texts and translations

Response Format

Body	Description
ID	String uniquely identifying the extension
csn	Compiled extension CSN
i18n	Texts and translations
timestamp	Timestamp of activation date

Example Request

Request Response

http
```http
PUT /-/cds/extensibility/Extensions/isbn-extension HTTP/1.1
Content-Type: application/json

{
  "csn": ["using my.bookshop.Books from '_base/db/data-model';
           extend my.bookshop.Books with { Z_ISBN: String };"],
  "i18n": [{ "name": "i18n.properties", "content": "Books_stock=Stock" },
            { "name": "i18n_de.properties", "content": "Books_stock=Bestand" }
}
```

The request can also be triggered asynchronously by setting the *`Prefer: respond-async`* header. You can use the URL returned in the *`Location`* response header to poll the job status.

In addition, you can poll the status for individual tenants using its individual task ID:

```http
GET /-/cds/jobs/pollTask(ID='<taskID>') HTTP/1.1
```

The response is similar to the following:

```js
{
  "status": "FINISHED",
  "op": "activateExtension"
}
```

The job and task status can take on the values *`QUEUED`*, *`RUNNING`*, *`FINISHED`* and *`FAILED`*.

> By convention, custom (tenant-specific) fields are usually prefixed with *`Z_`*.

The i18n data can also be passed in JSON format:

```json
"i18n": [{
  "name": "i18n.json",
  "content": "{\"\":{\"Books_stock\":\"Stock\"},\"de\":{\"Books_stock\":\"Bes
}]
```

You also get this JSON in the response body of PUT or GET requests. In this example, the text with key "Books_stock" from the base model is replaced.

DELETE *`Extensions/<ID>`*

Deletes a tenant-specific extension.

HTTP Request Options

Request Header	Example Value	Description
`prefer`	`respond-async`	Trigger asynchronous extension activation

Request Format

Parameters	Description
`ID`	String uniquely identifying the extension

Example Usage

```http
DELETE /-/cds/extensibility/Extensions/isbn-extension HTTP/1.1
Content-Type: application/json
```

The request can also be triggered asynchronously by setting the `Prefer: respond-async` header. You can use the URL returned in the `Location` response header to poll the job status.

In addition, you can poll the status for individual tenants using its individual task ID:

```http
GET /-/cds/jobs/pollTask(ID='<taskID>') HTTP/1.1
```

The response is similar to the following:

```js
{
  "status": "FINISHED",
  "op": "activateExtension"
}
```

The job and task status can take on the values `QUEUED` , `RUNNING` , `FINISHED` and `FAILED` .

Extension Restrictions

You can restrict what parts of the application model can be extended by the SaaS customer. This section lists the restrictions that you can define via the `extension-allowlist` .

> If an *`extension-allowlist`* is defined, only extensions in that list are allowed.

Using *`"for": ["*"]`* allows to apply rules to all entities and services.

Most checks happen at design time

`cds push` automatically builds the extension project, checking most restrictions locally. Some checks can only be performed at runtime, for example extension limit violations across multiple projects.

Restrict Service Extensions

By adding services to the *`extension-allowlist`*, services are enabled for extensions by Saas customers. In addition, you can restrict the number of bound entities by setting a limit for "new-entites".

```jsonc
"cds.xt.ExtensibilityService": {
  "extension-allowlist": [
    {
      // at most 2 new entities in CatalogService
      "for": ["CatalogService"],
      "new-entities": 2
    }
  ]
```

Restrict Entities and Fields

Entities can be extended with additional fields and also modifications of existing fields. Both kinds of extensions can be restricted.

- *`new-fields`* specifies the maximum number of fields that can be added.
- *`fields`* lists existing fields that are allowed to be extended. If the list is omitted, all fields can be extended.

```jsonc
"cds.xt.ExtensibilityService": {
  "extension-allowlist": [
    {
      // at most 2 new fields in entities from the my.bookshop namespace
      "for": ["my.bookshop"],
      "new-fields": 2,
      // allow extensions for field "description" only
```

```
        "fields": ["description"]
      },
      {
        // at most 1 new fields in my.bookshop.Authors
        "for": ["my.bookshop.Authors"],
        "new-fields": 1
      }
    ]
  }
```

This restriction allows two new fields for all entities in namespace *my.bookshop* but only one new field in entity *my.bookshop.Authors* .

The *field* restriction allows

```cds
extend my.bookshop.Books:description with (length: 2000);
```

but not, for example

```cds
extend my.bookshop.Books:title with (length: 200);
```

Restrict / Enable Annotations

The following annotations are blocked by default because they affect the persistence or security.

```txt
@restrict
@requires
@readonly
@mandatory
@assert.*
@cds.persistence.*
@sql.append
@sql.prepend
@path
@impl
@cds.autoexpose
@cds.api.ignore
@odata.etag
@cds.query.limit
@cds.localized
```

```
@cds.valid.*
@cds.search
```

You can, at your own risk, add exceptions for annotations.

```jsonc
"cds.xt.ExtensibilityService": {
      "extension-allowlist": [
        {
          "for": ["my.bookshop.Books"],
          "annotations": ["@mandatory", "@cds.api.ignore"]
        },
        {
          "for": ["my.bookshop.Authors:placeOfBirth"],
          "annotations": ["@mandatory"]
        }
      ]
  }
```

> Exception: *@cds.persistence.journal* cannot be applied as an extension to base entities.

Restrict Unbound Entities

You can also restrict unbound entities via their namespace.

For example

```jsonc
"cds.xt.ExtensibilityService": {
  "extension-allowlist": [
    {
      // at most 1 new entities for namepace my.new
      "for": ["my.new"],
      "new-entities": 1
    }
  ]
```

only allows one unbound entity with namespace *my.new* .

As a special case, you can also block any unbound entities:

```jsonc
"cds.xt.ExtensibilityService": {
  "extension-allowlist": [
    {
      // no new entities for all namespaces
      "for": ["*"],
      "new-entities": 0
    }
  ]
```

DeploymentService

The *DeploymentService* handles `subscribe` , `unsubscribe` , and `upgrade` events for single tenants and single apps or micro services. Actual implementation is provided through internal plugins, for example, for SAP HANA and SQLite.

Service Definition	*`@sap/cds-mtxs/srv/deployment-service`*
Service Definition Name	*`cds.xt.DeploymentService`*
Default HTTP Endpoint	*`/-/cds/deployment`*

Configuration

```jsonc
"cds.xt.DeploymentService": {
  "hdi": {
    "deploy": {
      ...
    },
    "create": {
      "database_id": "<SAP HANA Cloud instance ID>",
      ...
    },
    "bind": {
      ...
    }
  }
}
```

- Common Config Options

- `hdi` — bundles HDI-specific settings
  - `deploy` — HDI deployment parameters
  - `create` — tenant creation parameters (≈ `cf create-service`)
    - `database_id` — SAP HANA Cloud instance ID
  - `bind` — binding parameters (≈ `cf bind-service`)

Supported Presets

- `in-sidecar` — provides defaults for usage in sidecars
- `from-sidecar` — shortcut for `{ "kind": "rest" }`

`subscribe` *(tenant)*

Received when a new tenant subscribes.

The implementations create and initialize required resources, that is, creating and initializing tenant-specific HDI containers in case of SAP HANA, or tenant-specific databases in case of SQLite.

`upgrade` *(tenant)*

Used to upgrade a subscribed tenant.

Implementations read the latest models and content from the latest deployed version of the application and re-deploy that to the tenant's database.

Drop-Creating Databases for SQLite

In case of SQLite, especially in case of in-memory databases, an upgrade will simply drop and create a new tenant-specific database. Which means all data is lost.

Schema Evolution for SAP HANA

In case of SAP HANA, the delta to the former database layout will be determined, and corresponding CREATE TABLE, DROP-CREATE VIEW, and ALTER TABLE statements will eventually be executed without any data loss.

`unsubscribe` *(tenant)*

Received when a tenant is deleted.

The implementations free required resources, that is, dispose tenant-specific HDI containers in case of SAP HANA, or tenant-specific databases in case of SQLite.

SaasProvisioningService

The *SaasProvisioningService* is a façade for the *DeploymentService* to adapt to the API expected by SAP BTP's SaaS Provisioning service, hence providing out-of-the-box integration.

Service Definition	`@sap/cds-mtxs/srv/cf/saas-provisioning-service`
Service Definition Name	`cds.xt.SaasProvisioningService`
Default HTTP Endpoint	`/-/cds/saas-provisioning`

Configuration

jsonc

```jsonc
"cds.xt.SaasProvisioningService": {
  "jobs": {
    "queueSize": 5, // default: 100
    "workerSize": 5, // default: 1
    "clusterSize": 5, // default: 1
  }
}
```

- Common Config Options
- `jobs` — settings of the built-in job orchestrator
    - `workerSize` — max number of parallel asynchronous jobs per database
    - `clusterSize` — max number of database clusters, running `workerSize` jobs each
    - `queueSize` — max number of jobs waiting to run in the job queue

HTTP Request Options

Request Header	Example Value	Description
`prefer`	`respond-async`	Trigger subscription, upgrade or unsubscription request asynchronously.
`status_callback`	`/saas-manager/v1/subscription-callback/123456/result`	Callback path for SAP BTP SaaS Provisioning service. Set automatically if asynchronous subscription is configured for `saas-registry` service.

> **No `prefer: respond-async` needed with callback**
>
> Requests are implicitly asynchronous when `status_callback` is set.

Example Usage

Subscription

A subscription for a tenant `t1` with a specific database ID:

```http
PUT /-/cds/saas-provisioning/tenant/t1 HTTP/1.1
Content-Type: application/json

{
  "subscribedTenantId": "t1",
  "eventType": "CREATE",
  "_": {
    "hdi": {
      "create": {
        "database_id": "<SAP HANA Cloud instance ID>"
      }
    }
  }
}
```

Upgrade

With `@sap/hdi-deploy` parameters `trace` and `version`:

```http
POST /-/cds/saas-provisioning/upgrade HTTP/1.1
Content-Type: application/json
```

```
{
    "tenants": ["t1"],
    "options": {
        "_": {
            "hdi": {
                "deploy": {
                    "trace": "true",
                    "version": "true"
                }
            }
        }
    }
}
```

GET *tenant/<tenant>*

Returns tenant-specific metadata if `<tenant>` is set, and a list of all tenants' metadata if omitted.

Parameters	Description
tenant	A string identifying the tenant.

Example Usage

Get Metadata for a Specific Tenant

Request Response

http

```
GET /-/cds/saas-provisioning/tenant/t1 HTTP/1.1
Content-Type: application/json
```

Get Metadata for All Tenants

Request Response

http

```
GET /-/cds/saas-provisioning/tenant HTTP/1.1
Content-Type: application/json
```

PUT *tenant/<tenant>* (...)

Creates tenant resources required for onboarding.

Learn about query parameters, arguments, and their description in the following table:

Parameters	
tenant	A string identifying the tenant
Arguments	
subscribedTenantId	A string identifying the tenant
subscribedSubdomain	A string identifying the tenant-specific subdomain
eventType	The *saas-registry* event (*CREATE* or *UPDATE*)

Example Usage

Request Response

http
```http
PUT /-/cds/saas-provisioning/tenant/t1 HTTP/1.1
Content-Type: application/json

{
  "subscribedTenantId": "t1",
  "subscribedSubdomain": "subdomain1",
  "eventType": "CREATE"
}
```

DELETE *tenant/<tenant>*

Deletes all tenant resources.

GET *dependencies* → *[{ xsappname }]*

Returns configured SAP BTP SaaS Provisioning service dependencies.

↳ *Learn how to configure SaaS dependencies*

`upgrade` *[tenants] → Jobs*

Use the *`upgrade`* endpoint to upgrade tenant base models.

Arguments	Description
`tenants`	A list of tenants, or *`[*]`* for all tenants
`options`	Additional options, including HDI deployment options, see DeploymentService, prefixed with `_`

Example Usage

Asynchronously Upgrade a List of Tenants

Request Response

http
```
POST /-/cds/saas-provisioning/upgrade HTTP/1.1
Content-Type: application/json
Prefer: respond-async

{
  "tenants": ["t1", "t2"],
  "options": {
      "_": {
          "hdi": {
              "deploy": {
                  "trace": "true",
                  "version": "true"
              }
          }
      }
    }
}
```

Asynchronously Upgrade All Tenants

Request Response

http
```
POST /-/cds/saas-provisioning/upgrade HTTP/1.1
Content-Type: application/json
Prefer: respond-async

{
  "tenants": ["*"]
}
```

We recommended to execute the upgrades asynchronously by setting the `Prefer: respond-async` header. You can use the URL returned in the `Location` response header to poll the job status.

In addition, you can poll the status for individual tenants using its individual task ID:

```http
GET /-/cds/jobs/pollTask(ID='<taskID>') HTTP/1.1
```

The response is similar to the following:

```js
{
  "status": "FINISHED",
  "op": "upgrade"
}
```

The job and task status can take on the values `QUEUED`, `RUNNING`, `FINISHED` and `FAILED`.

Old MTX Reference

↳ *See Reference docs for former 'old' MTX Services.*