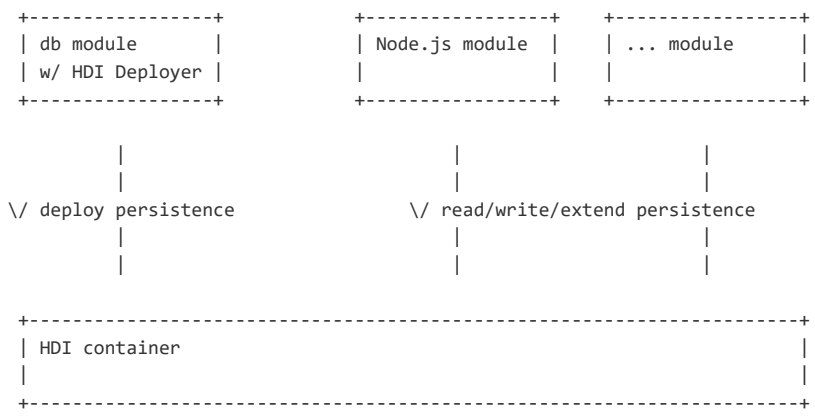
stable: master - change log - latest release | development: devel - change log - all releases - development guidelines

sap-hdi-deploy

This is the Node.js-based HANA DI (HDI) Deployer for XS Advanced (XSA).

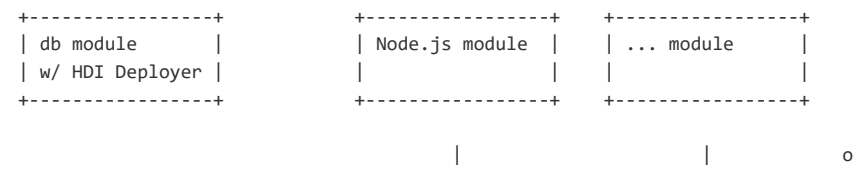
The HDI Deployer is an application that is included in an Multi-Target Application (MTA) and is used to deploy HDI design-time artifacts to the respective HDI containers. When an MTA is deployed, the HDI Deployer application is pushed first so that it can "prepare" the SAP HANA persistence; by the time defined services are started in XS Advanced, the HDI container is ready for use.

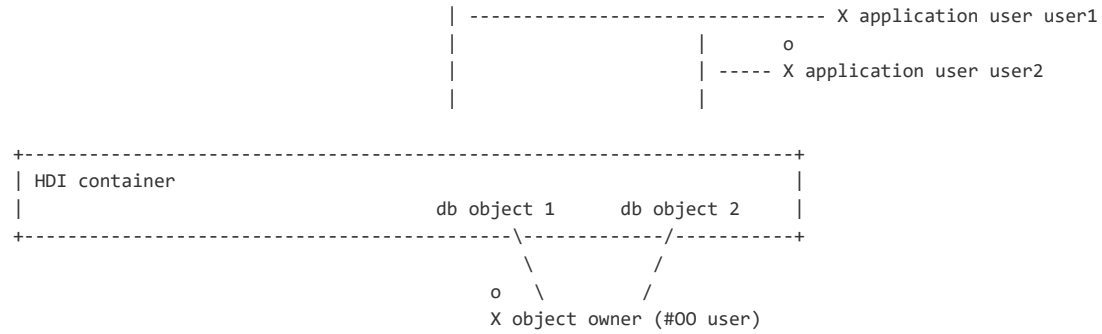
For an MTA with different modules, e.g. a db module, a Node.js module, etc., this looks as follows:



In the XSA HDI setup, each module of the MTA is equipped with it's own technical database user for accessing the runtime schema of the HDI container.

The following diagram illustrates the different users who are involved in this setup with regard to privileges: the application users user1 and user2 who are bound to one of the XSA modules each, and the HDI container's object owner (the #OO user) who is the owner of the objects in the database persistence of the MTA which are managed by HDI:





The HDI Deployer is packaged into the `db` module of the MTA. So, in order to use a new HDI Deployer, you need to reference a new version of the HDI Deployer in the `db` module's `package.json` file.

The HDI Deployer supports HANA 1.0 SPS11, HANA 1.0 SPS12, and the current HANA 2.0 development codeline.

Note: The HDI Deployer assumes ownership of the `src/`, `cfg/`, and `lib/` folders in the bound target HDI container. Binding more than 1 instance of the HDI Deployer to the same HDI container as the target container, e.g. the `db` modules of 2 MTAs are bound to the same HDI container as the target container, is not supported and results in undefined behavior.

README.md

Installation:

The XSA Database Module:

Configuration and Reconfiguration:

Configuration of NPM

In order to install `sap-hdi-deploy`, ensure that `npm` is configured appropriately:

```
npm config set registry "http://nexus.wdf.sap.corp:8081/nexus/content/groups/build.milestones.npm/"
npm config rm proxy
npm config rm https-proxy
npm config set strict-ssl false
```

Integration into an XSA Database Module

Usually, `sap-hdi-deploy` gets installed via a `package.json`-based dependency inside your XSA's application's `db` module:

`db/package.json`:

```
{
  "name": "deploy",
  "dependencies": {
    "sap-hdi-deploy": "2.2.0"
  },
  "scripts": {
    "start": "node node_modules/sap-hdi-deploy/deploy.js"
  }
}
```

`sap-hdi-deploy` will be downloaded and installed via `npm`:

```
npm install
```

Standalone Download and Installation

`sap-hdi-deploy` can be downloaded and installed as a normal application directly from GitHub:

```
git clone https://github.wdf.sap.corp/xs2/hdideploy.js.git
npm install hdideploy.js/
```

Database Connection Details

Connection details for the database, e.g. host, port, credentials, and certificates, are looked up by the HDI Deployer from the standard CF/XSA `VCAP_SERVICES` environment variable which contains the bound services.

For local testing, the HDI Deployer supports default configurations via the following configuration files:

- `default-services.json`: a JSON file which contains a set of service bindings
- `default-env.json`: a JSON file which contains a set of environment variables and their values

A Database Module's File System Structure

The HDI Deployer expects the following file system structure for your the HDI content in your `db` module:

- `src/`: folder which contains your HDI source artifacts
- `cfg/`: optional folder with HDI configuration artifacts
- `package.json`: this file is used by npm (the Node.js package manager) to bootstrap and start the application

Other files in the root directory will be ignored by `sap-hdi-deploy`.

Delta Deployment and Undeploy Whitelist

The HDI Deployer implements a delta-based deployment strategy:

On startup, the HDI Deployer recursively scans the local `src/` and `cfg/` folders, processes config templates, looks at the HDI container at the server-side and calculates the set of added, modified, and deleted files based on the difference between the local file system state and the deployed file system state of the server-side HDI container.

In normal operation, the HDI Deployer will schedule only the set of added and modified files to deployment. The set of deleted files is not scheduled for undeployment.

In order to undeploy deleted files, an application needs to include an undeploy whitelist via an `undeploy.json` file in the root directory of the `db` module (right beside the `src/` and `cfg/` folders). The undeploy whitelist `undeploy.json` file is a JSON document with a top-level array of file names:

`undeploy.json`:

```
[
    "src/Table.hdbcds",
    "src/Procedure.hdbprocedure"
]
```

The file must list all artifacts which should be undeployed. The file path of the artifacts must be relative to the root directory of the `db` module and must use the HDI file path delimiter '/'.

For interactive scenarios, it's possible to pass the `auto-undeploy` option to the HDI Deployer, e.g.

```
node deploy --auto-undeploy
```

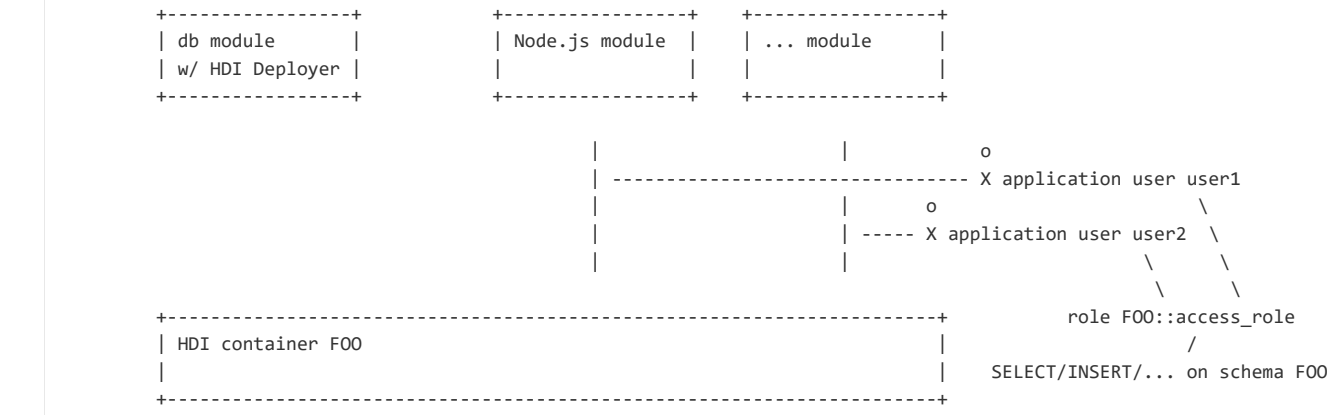
In this case, the HDI Deployer will ignore the undeploy whitelist `undeploy.json` file and will schedule all deleted files in the `src/` and `cfg/` folders for undeployment.

The default_access_role Role

When an HDI container service instance is created by the HANA Service Broker, e.g. service instance `foo` with schema name `FOO`, the broker creates an HDI container `FOO` (consisting of the runtime schema `FOO`, the HDI metadata and API schema `FOO#DI`, and the object owner `FOO#OO`) and a global access role `FOO::access_role` for the runtime schema. This access role is equipped with a default permission set for the runtime schema which consists of `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE`, `CREATE TEMPORARY TABLE`, and `SELECT CDS METADATA` on the runtime schema `FOO`.

Every time the service instance is bound to an application, the broker creates 2 new users which are specific to this binding. The first user is the application user who is named `user` in the instance's credentials. This user is used by the application to access the HDI container's runtime schema `FOO`. This user is equipped with the service instance's global access role `FOO::access_role`. The second user is the HDI API user who is named `hdi_user` in the credentials. This user is equipped with privileges for the container's APIs in the `FOO#DI` schema.

The following diagram illustrates the binding-specific application users and the role of the global access role (the HDI API users and the bindings for the HDI Deployer are not shown for simplicity):



Exemplary service binding:

```
{
   "hana" : [ {
```

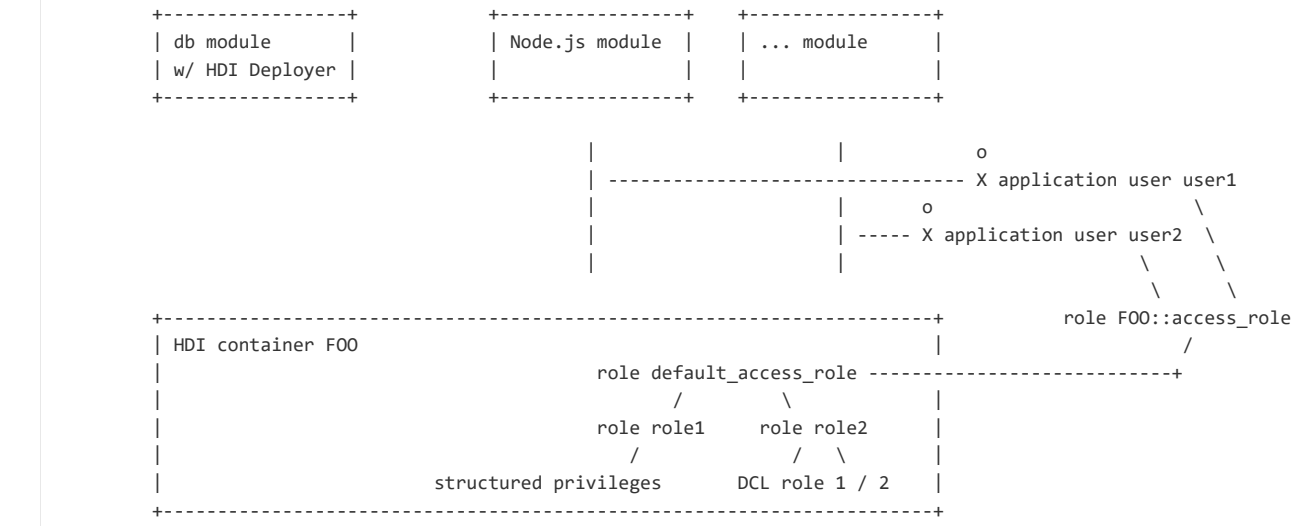
```
      "name" : "foo",
      "label" : "hana",
      "tags" : [ "hana", "database", "relational" ],
      "plan" : "hdi-shared",
      "credentials" : {
        "schema" : "FOO",
        "driver" : "com.sap.db.jdbc.Driver",
        "port" : "30115",
        "host" : "srv1234567.host.name",
        "db_hosts" : [ {
          "port" : 30115,
          "host" : "srv1234567.host.name"
        } ],
        "user" : "SBSS_345999596729021957418757608738537665554047278221560600568361494 75",
        "password" : "<password>",
        "hdi_user" : "SBSS_64592794580116217572062412157356606994624009694957290675610125954",
        "hdi_password" : "<password>",
        "url" : "jdbc:sap://srv1234567.host.name:30115/?currentschema=FOO"
      }
    } ]
}
```

In order to assign roles from the HDI content to the application binding users (the `user` users), the HDI Deployer implements an automatic assignment of the `default_access_role` role if it is present in the deployed content:

If a role definition file exists at the path `src/defaults/default_access_role.hdbrole`, and this file defines a role named `default_access_role`, and this file is included in the deployment (e.g. not excluded via `include-filter`), then the HDI Deployer grants the deployed `default_access_role` role to the service instance's global access role (e.g. `FOO::access_role`). In addition, the HDI Deployer revokes all default permissions (e.g. `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `EXECUTE`, `CREATE TEMPORARY TABLE`, and `SELECT CDS METADATA` on the runtime schema `FOO`) from the global access role.

Note: If you use a `.hdinamespace` file in `src/` which defines a real namespace prefix for subfolders, then you need to put a `.hdinamespace` file with the empty namespace `"name" : ""` at `src/defaults/` to ensure that the role can be named `default_access_role`.

The following diagram illustrates the binding-specific application users, the role of the global access role, and the container-specific default access role:



Note: The `default_access_role` is assumed to be an "umbrella" role which aggregates other roles.

Reusable Database Modules

In order to allow that an application uses (parts of) the database persistence of a reusable component inside its own persistence model, the HDI Deployer allows to link/include the design-time files of reusable components in a consuming application in an automated way. This mechanism is based on the Node.js package management mechanism for defining, publishing, and consuming reusable database modules which also supports versioning based on the semantic versioning concepts (cf. http://semver.org).

A reusable database module is considered to have the same `src/` and `cfg/` folder structure as a normal database module. The `src/.hdiconfig` file is mandatory and used by the module mechanism as an indicator that the `src/` and `cfg/` folders belong to a consumable, reusable database module. In addition, the reusable database module needs to have a `package.json` file which defines the module's name, the module's version, etc.

A complete reusable database module would looks as follows:

```
/
+-- src/
|   +-- .hdiconfig
|   +-- <source files ...>
+-- cfg/
|   +-- <optional configuration files ...>
+-- package.json
```

The `package.json` file contains the module's name, description, version, repository URL, and the set of files which belong to the module:

`package.json`:

```
{
  "name": "module1",
  "description": "A set of reusable database objects",
  "version": "1.3.1",
  "repository": {
    "url": "git@github.com:modules/module1.git"
  },
  "files": [
    "src",
    "cfg",
    "package.json"
  ]
}
```

The reusable database module should be published to a Node.js package management compliant object repository.

Consumption of a reusable database module is done by adding a dependency in the consuming module's `package.json` file, right beside the dependency to `sap-hdi-deploy`:

```
{
  "name": "deploy",
  "dependencies": {
    "sap-hdi-deploy": "2.2.0",
    "module1": "1.3.1",
    "module2": "1.7.0"
  },
  "scripts": {
    "start": "node node_modules/sap-hdi-deploy/deploy.js"
  }
}
```

Here, the consuming module requires `module1` in version `1.3.1` and `module2` in version `1.7.0`.

When running `npm install` to download and install the dependencies which are listed in the dependencies section of the `package.json` file, `npm` will also download the reusable database modules and places them into the `node_modules/` folder of

the consuming module. For each module a separate subfolder is created with the name of the module.

When the HDI Deployer is triggered to do the actual deployment of the (consuming) database module, it scans the `node_modules/` folder and virtually integrates the `src/` and `cfg/` folders of found reusable database modules into the (consuming) database module's `lib/` folder. Reusable database modules are identified by the mandatory `src/.hdiconfig` file.

On successful deployment, the HDI container will contain the consumed modules below the root-level `lib/` folder, e.g.

```
/
+-- src/
+-- cfg/
+-- lib/
|   +-- module1/
|   |   +-- src/
|   |   +-- cfg/
|   +-- module2/
|       +-- src/
|       +-- cfg/
```

For the time being, it's not allowed to recursively include reusable database modules.

The `cfg/` folders of reusable database modules are also subject to configuration file templating.

Configuration File Templating

The HDI Deployer implements a templating mechanism for HDI configuration files, e.g. configuration files for synonyms, projection views, etc., based on services which are bound to the `db` module application. By means of this templating mechanism, it is possible to configure synonyms, projection views, etc. to point to the right database schema without knowing the schema name at development time.

On startup, the HDI Deployer recursively scans the local `cfg/` folder and picks up all files with a `.*config` suffix, e.g. all `.hdbsynonymconfig`, `.hdbvirtualtableconfig`, etc. files. For all collected files which contain `.configure` markers in their content, it applies the configuration templating and creates transient configuration files which are then deployed to the HDI container.

For a synonym configuration file `cfg/LOCAL_TARGET.hdbsynonymconfig`

```
{
    "LOCAL_TARGET" : {
        "target" : {
            "schema.configure"    : "logical-external-service/schema",
            "database.configure"  : "logical-external-service/database",
            "object"              : "TARGET"
        }
    }
}
```

the section

```
            "schema.configure"    : "logical-external-service/schema",
            "database.configure"  : "logical-external-service/database",
            "object"              : "TARGET"
```

will be transformed by the templating mechanism into

```
            "schema"              : "THE_SCHEMA",
            "database"            : "THE_DATABASE",
            "object"              : "TARGET"
```

where `THE_SCHEMA` and `THE_DATABASE` are the values for the `schema` and `database` fields of the bound service `logical-external-service`, which are denoted by the path expressions `logical-external-service/schema` and `logical-external-service/database`.

If a field in the service is missing, it will not be configured and will be removed instead, e.g. `database` might be optional.

The names of the services are subject to the service replacements mechanism, which can be used to map a real service, e.g. `real-external-service`, to a logical service name which is used in the configuration files, e.g. `logical-external-service`.

It's not always applicable to use `schema.configure`, `database.configure`, etc. in the configuration template files. Therefore, the HDI Deployer provides a generic way of copying a set of properties from the bound service, e.g. schema, database, remote source, etc. if they are present, although the template file doesn't mention them.

For the configuration file `cfg/LOCAL_TARGET.hdbsynonymconfig` this could looks as follows:

```
{
    "LOCAL_TARGET" : {
        "target" : {
            "*.configure"         : "logical-external-service",
            "object"              : "TARGET"
        }
    }
}
```

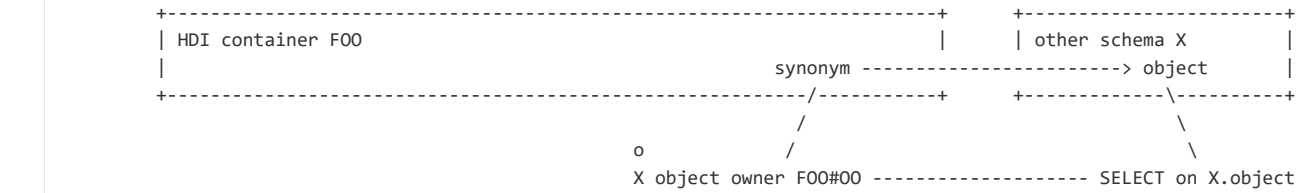
When the HDI Deployer encounters a `*.configure` entry, it simply copies all well-known fields which are present in the bound service into the configuration file. The well-known fields are currently `remote`, `database`, and `schema`.

The HDI Deployer also supports old-style `.hdbsynonymtemplate` template files: If a `.hdbsynonymtemplate` file is found in the `cfg/` or `src/` folder, then it is processed as a configuration template file and a transient file with the suffix `.hdbsynonymconfig` is created. A field `grantor` is replaced with the `schema` value from the referenced service; so, a `grantor` field is equivalent to a `"schema.configure" : "the-service/schema"` entry in a configuration template file.

Permissions to Container-External Objects

An HDI container is by default equipped with (nearly) zero database privileges, e.g. the object owner (`#OO` user) is mainly equipped with the `CREATE ANY` privilege on the container's runtime schema (e.g. schema `FOO` for an HDI contaner `FOO`).

In order to access database objects inside other database schemata or other HDI containers, and in order to deploy synonyms into the HDI container which point to these container-external objects, at least the object owner needs additional privileges, e.g. for an object `object` in schema `X` `SELECT` privileges on `X.object`:



In order to automatically assign privileges to the object owner and/or the application binding users, the HDI Deployer provides `.hdbgrants` files with a syntax similar to `.hdbrole` files:

An `.hdbgrants` file has the following structure:

`granting-service.hdbgrants`:

```
{
  "granting-service": {
```

```
    "object_owner": {
      <privileges>
    },
    "application_user": {
      <privileges>
    }
  }
}
```

The top-level keys define the names of the bound services which "grant" the privileges, these are the "grantors", e.g. `granting-service` in the example. The next level defines to which users the privileges will be granted, these are the "grantees": `object_owner` is used for the HDI container's object owner, and `application_user` marks the application users which are bound to the application modules, e.g. the Node.js module. The third level defines the set of privileges in a `.hdbrole` -like structure.

On startup, the HDI Deployer looks for `.hdbgrants` files and processes them as follows: For each grantor in the file, the HDI Deployer looks up a bound service with the name (subject to service replacements), connects to the database with the service's credentials, and grants the specified privileges to the grantees.

For backwards compatibility, also the suffix `.hdbsynonymgrantor` is supported.

Example of a `cfg/SYS-access.hdbgrants` file with some privileges:

```
{
  "SYS-access": {
    "object_owner": {
      "object_privileges":[
        {
          "schema": "SYS",
          "name": "VIEWS",
          "privileges": ["SELECT"]
        },
        {
          "schema": "SYS",
          "name": "TABLES",
          "privileges": ["SELECT"]
        }
      ]
    },
    "application_user": {
      "object_privileges":[
        {
          "schema": "SYS",
          "name": "VIEWS",
          "privileges": ["SELECT"]
        },
        {
          "schema": "SYS",
          "name": "TABLES",
          "privileges": ["SELECT"]
        }
      ]
    }
  }
}
```

Grant permissions on target of synonym

Note: This section needs to be refurbished.

Privileges

To grant an user privileges of database objects, which do not belong to the user's container, one or more .hdbsynonymgrantor files has to be provided as additional deployment content.

Example

- .hdbsynonymgrantor files are in JSON format with a syntax similar to .hdbrole files
- the JSON (top-level) 'object keys' are the name of a service binding
- the privileges listed in the 'container' section will be granted to the HDI container object owner
- the privileges listed in the 'user' section will be granted to all container users via roles

Privileges are granted to the container object owner and the container users as follows:

- for each grantor object defined in a .hdisynonymgrantor file the deploy app connects to the database with the user credentials given by the service binding and
- grants the privileges to the user of the target container (see target container)
- in case of an error (grantor user is not privileged to grant privilege to container user or the object to be granted doesn't exist), the deploy app provide an error protocol and stops execution

Target schema for 'object_privileges':

- can be defined as schema attribute in the 'object_privileges' object definition or
- will be taken from the service binding

In case no schema definition is available (e.g. user defined service without schema in credentials) the schema part for privilege granting will be omitted.

Cross HDI container privileges

HDI container object privileges can only be granted to other containers via container local roles. Please follow these steps to grant object privileges of a 'grantor container' to application users of a 'grantee container':

- deploy one or more .hdbrole files defining object privileges to the 'grantor container'
- reference these roles in the 'container_roles' sections of a .hdbsynonymgrantor file for 'grantee container' deployment

Example

Target container

If more than 1 HANA database service (e.g. a grantor-service and a hdi-container) is bound to the HDI Deployer app, then you need to specify the TARGET_CONTAINER property.

Example:

```
applications:
- name: events-db
  path: db
  no-route: true
  services:
    - events-database
    - grantor-erp
    - othercontainer
  env:
    TARGET_CONTAINER: events-database
```

Synonym Definition / Configuration

The DeployApp does not interpret or manipulate any of the .hdbsynonym or .hdbsynonymconfig files. To ease the target schema definition a .hdbsynonymtemplate can be provided instead of a .hdisynonymconfig file.

The .hdbsynonymtemplate format is similar to .hdbsynonymconfig except having the schema attribute replaced with a grantor attribute:

Example

- the deploy app writes transiently a .hdbsynonymconfig for all .hdbsynonymtemplate files replacing the grantor attribute with the target schema
- for schema replacements the same rules will be applied as described for .hdbsynonymgrantor

Examples

Grant non-container privileges

Please follow these steps to grant privileges to a container for non-container database objects:

- create an user defined service tagged with 'hana'

```
cf|xs cups <user-defined-service> -p "{\"host\":\"<hana host>\",\"port\":\"<hana port\",\"user\":\"<user>\",\"passwor
```

- set the TARGET_CONTAINER environment variable in the manifest.yml
- place a .hdbsynonymgrantor file in the db/cfg directory
- place synonym definition files (.hdbsynonym and .hdbsynonymtemplate) in the db/src directory

Example - grant privilege on SYS.USERS to target container synonym Syn.Users

Grant cross container table privileges

node-hello-world-xcntsyn is a version of the node-hello-world application referencing the AddressBook entities via synonyms.

To run it, please follow these steps:

- add the AddressBook.hdbrole file to the node hello world db/src directory
- deploy (push) the node-hello-world application
- maintain address books and corresponding adresses with the node-hello-world application
- create service instance node-hdi-container-xcntsyn and deploy (push) the node-hello-world-xcntsyn application
- verify, that the address data maintained in node-hello-world is visible in the node-hello-world-xcntsyn application

Grant cross container table privileges to database view via synonyms

node-hello-world-xcntviewsyn is a version of the node-hello-world application referencing the AddressBook entities via a database view on basis of synonyms.

To run it, please follow these steps:

- add the AddressBookView.hdbrole file to the node hello world db/src directory
- deploy (push) the node-hello-world application
- maintain address books and corresponding adresses with the node-hello-world application
- create service instance node-hdi-container-xcntviewsyn and deploy (push) the node-hello-world-xcntviewsyn application
- verify, that the address data maintained in node-hello-world is visible in the node-hello-world-xcntviewsyn application

Environment Variables for Applications

`sap-hdi-deploy` supports (re-)configuration via the following environment variables which are exposed to applications, e.g. via the CF/XSA `manifest.yml` or the MTA descriptor `mta.yaml` :

- `TARGET_CONTAINER` : (optional) service name that specifies the HDI target container (needed, if more than one service is bound to the HDI Deployer)
- `SERVICE_REPLACEMENTS` : (optional) JSON-structured list of service replacements, e.g. `[ { "key": "logical-service-name-1", "service":"real-service-name-1"}, { "key": "logical-service-name-2", "service":"real-service-name-2"} ]` , where the logical service names refer to the names in the HDI content and the real service names refer to the services

which are bound to the HDI Deployer via `VCAP_SERVICES` ; if the HDI content references a service name which is not listed in the replacements, then this name is used as a real service name

The structure of the `SERVICE_REPLACEMENTS` environment variable is based on the MTA specification (page 13f) in order to enable MTA group assignments.

Example `manifest.yml` :

```
applications:
- name: app-db
  path: db
  services:
    - app-database
    - real-grantor-service
    - real-external-service
  env:
    TARGET_CONTAINER: app-database
    SERVICE_REPLACEMENTS: >
    [
      {
        "key"     : "logical-grantor-service",
        "service" : "real-grantor-service"
      },
      {
        "key"     : "logical-external-service",
        "service" : "real-external-service"
      }
    ]
```

Environment Variables for Infrastructure / Development Tools

`sap-hdi-deploy` supports (re-)configuration via the following environment variables for infrastructure / development tools like the XSA Deploy Service or internal build tools of the WEB IDE

- `DEPLOY_ID` : (optional) if set, the given id will be written to the final application log entry (custom id, to support processes in parsing log output
- `HDI_DEPLOY_OPTIONS` : (optional) JSON-structured set of options for the HDI Deployer, e.g. `{ "auto_undeploy" : true, "exit" : true, "root" : "/volumes/A/workspaces/B/db/", "include_filter" : [ "src/", "cfg/" ] }`

Options for Interactive Scenarios

`sap-hdi-deploy` supports the following options for interactive deployment scenarios, e.g. for orchestration via the WEB IDE or for CI scripts:

- `--[no-]verbose` : [don't] print detailed log messages to the console
- `--structured-log <file>` : write log messages as JSON objects into the given file; messages are appended if the file already exists
- `--[no-]exit` : [don't] exit after deployment of artifacts
- `--root <path>` : use the given root path for artifacts
- `--include-filter [<path> ..]` : only include the given paths (directories and files) when processing files
- `--deploy [<file> ..]` : explicitly schedule the given files for deploy; extends the `include-filter` for collecting local files
- `--undeploy [<file> ..]` : explicitly schedule the given files for undeploy
- `--[no-]auto-undeploy` : [don't] undeploy artifacts automatically based on delta detection and ignore the `undeploy.json` file
- `--[no-]treat-warnings-as-errors` : [don't] treat warnings as errors
- `--[no-]simulate-make` : [don't] simulate the make and skip post-make activities; pre-make activities still take effect, e.g. grants

See `--help` for details and defaults.

Options can also be passed to `sap-hdi-deploy` via the `HDI_DEPLOY_OPTIONS` environment variable.

Supported Features

`sap-hdi-deploy` exposes its set of features via the `info` option, which can be passed as `--option` or via `HDI_DEPLOY_OPTIONS`, e.g.

```
node deploy --info [<component> [<component> [...]]]
```

where a list of components can be specified.

The `info` option allows to pass multiple components. The `info` request for these components is optional, e.g. if the HDI Deployer doesn't support the component, then it will not throw an error, but simply not return information for that component. The special component `all` will return the information for all known components; `all` is the default if no component is specified. For certain future components, e.g. `server`, the HDI Deployer might need to connect to the HDI container in the database and retrieve feature information from there.

Examples:

```
node deploy --info all
node deploy --info client server
```

The result of an `info` call is a JSON document where the top-level objects correspond to the requested components. Each component should at least report its name, its version, and the set of supported features with name and version number (version numbers are simple numbers (no dots, no double-dots)).

If a version number is negative, then the feature is supported by the client, but not supported by the server.

For a `--info client` call, the document looks as follows:

```
{
    "client": {
        "name": "sap-hdi-deploy",
        "version": "2.2.0",
        "features": {
            "info": 2,
            "verbose": 1,
            "structured-log": 1,
            "default-access-role": 1,
            "grants": 2,
            "include-filter": 1,
            "deploy": 1,
            "undeploy": 1,
            "treat-warnings-as-errors": 1,
            "simulate-make": 1,
            "service-replacements": 1,
            "modules": 1,
            "config-templates": 2,
            "environment-options": 1,
            "undeploy-whitelist": 1
        }
    },
    "server": {
        "name": "sap-hana-database",
        "version": "1.00.120.04.0000000000",
        "features": {}
    }
}
```