Using PostgreSQL

This guide focuses on the new PostgreSQL Service provided through *@cap-js/postgres* , which is based on the same new database services architecture as the new SQLite Service. This architecture brings significantly enhanced feature sets and feature parity, as documented in the *Features* section of the SQLite guide.

↳ *Learn about migrating from the former `cds-pg` in the Migration chapter.*

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

Table of Contents

Setup & Configuration

Run this to use PostgreSQL for production:

```sh
npm add @cap-js/postgres
```

Auto-Wired Configuration

The *@cap-js/postgres* package uses *cds-plugin* technique to auto-configure your application and use a PostgreSQL database for production.

You can inspect the effective configuration using *cds env* :

```sh
cds env requires.db --for production
```

Output:

```js
{
  impl: '@cap-js/postgres',
  dialect: 'postgres',
  kind: 'postgres'
}
```

↳ *See also the general information on installing database packages*

Provisioning a DB Instance

To connect to a PostgreSQL offering from the cloud provider in Production, leverage the PostgreSQL on SAP BTP, hyperscaler option . For local development and testing convenience, you can run PostgreSQL in a docker container.

Using Docker

You can use Docker to run a PostgreSQL database locally as follows:

1. Install and run Docker Desktop
2. Create the following file in your project root directory:

pg.yml

```yaml
services:
  db:
    image: postgres:alpine
    environment: { POSTGRES_PASSWORD: postgres }
    ports: [ '5432:5432' ]
    restart: always
```

3. Create and run the docker container:

```sh
docker-compose -f pg.yml up -d
```

Service Bindings

You need a service binding to connect to the PostgreSQL database.

In the cloud, use given techniques to bind a cloud-based instance of PostgreSQL to your application.

For local development provide the credentials using a suitable `cds env` technique, like one of the following.

Configure Service Bindings

Using Defaults with *[pg]* Profile

The *@cds-js/postgres* comes with default credentials under profile *[pg]* that match the defaults used in the docker setup. So, in case you stick to these defaults you can skip the next sections and just go ahead, deploy your database:

```sh
cds deploy --profile pg
```

Run your application:

```sh
cds watch --profile pg
```

↳ *Learn more about that in the Deployment chapter below.*

In Your private *~/.cdsrc.json*

Add it to your private *~/.cdsrc.json* if you want to use these credentials on your local machine only:

~/.cdsrc.json

```json
{
  "requires": {
    "db": {
      "[pg]": {
        "kind": "postgres",
        "credentials": {
          "host": "localhost", "port": 5432,
          "user": "postgres",
          "password": "postgres",
          "database": "postgres"
        }
      }
    }
  }
}
```

In Project *.env* Files

Alternatively, use a *.env* file in your project's root folder if you want to share the same credentials with your team:

.env

```properties
cds.requires.db.[pg].kind = postgres
cds.requires.db.[pg].credentials.host = localhost
cds.requires.db.[pg].credentials.port = 5432
cds.requires.db.[pg].credentials.user = postgres
cds.requires.db.[pg].credentials.password = postgres
cds.requires.db.[pg].credentials.database = postgres
```

Using Profiles

The previous configuration examples use the *cds.env* profile *[pg]* to allow selectively testing with PostgreSQL databases from the command line as follows:

```sh
cds watch --profile pg
```

The profile name can be freely chosen, of course.

Deployment

Using *cds deploy*

Deploy your database as usual with that:

```sh
cds deploy
```

Or with that if you used profile *[pg]* as introduced in the setup chapter above:

```sh
cds deploy --profile pg
```

With a Deployer App

When deploying to Cloud Foundry, this can be accomplished by providing a simple deployer app. Similar to SAP HANA deployer apps, it is auto-generated for PostgreSQL-enabled projects by running

```sh
cds build --production
```

▶ *What `cds build` does...*

Add PostgreSQL Deployment Configuration

```sh
cds add postgres
```

▶ *See what this does...*

Deploy

You can package and deploy that application, for example using MTA-based deployment.

Automatic Schema Evolution

When redeploying after you changed your CDS models, like adding fields, automatic schema evolution is applied. Whenever you run `cds deploy` (or `cds-deploy`) it executes these steps:

1. Read a CSN of a former deployment from table `cds_model`.
2. Calculate the **delta** to current model.
3. Generate and run DDL statements with:
   - `CREATE TABLE` statements for new entities
   - `CREATE VIEW` statements for new views
   - `ALTER TABLE` statements for entities with new or changed elements

    - `DROP & CREATE VIEW` statements for views affected by changed entities
4. Fill in initial data from provided *.csv* files using `UPSERT` commands.
5. Store a CSN representation of the current model in `cds_model`.

> You can disable automatic schema evolution, if necessary, by setting `cds.requires.db.schema_evolution = false` ⚙.

No manual altering

Manually altering the database will most likely break automatic schema evolution!

Limitations

Automatic schema evolution only allows changes without potential data loss.

Allowed

- Adding entities and elements
- Increasing the length of Strings
- Increasing the size of Integers

Disallowed

- Removing entities or elements
- Changes to primary keys
- All other type changes

For example the following type changes are allowed:

```cds
entity Foo {
   anInteger : Int64;     // from former: Int32
   aString : String(22);  // from former: String(11)
}
```

TIP

> If you need to apply such disallowed changes during development, just drop and re-create your database, for example by killing it in docker and re-create it using the `docker-compose` command, see Using Docker.

Dry-Run Offline

You can use `cds deploy` with option `--dry` to simulate and inspect how things work.

1. Capture your current model in a CSN file:

   sh
   ```sh
   cds deploy --dry --model-only --out cds-model.csn
   ```

2. Change your models, for example in *capire/bookshop/db/schema.cds* :

   cds
   ```cds
   entity Books { ...
      title : localized String(222); //> increase length from 111 to 222
      foo : Association to Foo;      //> add a new relationship
      bar : String;                  //> add a new element
   }
   entity Foo { key ID: UUID }       //> add a new entity
   ```

3. Generate delta DDL statements:

   sh
   ```sh
   cds deploy --dry --delta-from cds-model.csn --out delta.sql
   ```

4. Inspect the generated SQL statements, which should look like this:

   delta.sql

   sql
   ```sql
   -- Drop Affected Views
   DROP VIEW localized_CatalogService_ListOfBooks;
   DROP VIEW localized_CatalogService_Books;
   DROP VIEW localized_AdminService_Books;
   DROP VIEW CatalogService_ListOfBooks;
   DROP VIEW localized_sap_capire_bookshop_Books;
   DROP VIEW CatalogService_Books_texts;
   DROP VIEW AdminService_Books_texts;
   DROP VIEW CatalogService_Books;
   DROP VIEW AdminService_Books;
   ```

```
-- Alter Tables for New or Altered Columns
ALTER TABLE sap_capire_bookshop_Books ALTER title TYPE VARCHAR(222);
ALTER TABLE sap_capire_bookshop_Books_texts ALTER title TYPE VARCHAR(222)
ALTER TABLE sap_capire_bookshop_Books ADD foo_ID VARCHAR(36);
ALTER TABLE sap_capire_bookshop_Books ADD bar VARCHAR(255);

-- Create New Tables
CREATE TABLE sap_capire_bookshop_Foo (
  ID VARCHAR(36) NOT NULL,
  PRIMARY KEY(ID)
);

-- Re-Create Affected Views
CREATE VIEW AdminService_Books AS SELECT ... FROM sap_capire_bookshop_Boo
CREATE VIEW CatalogService_Books AS SELECT ... FROM sap_capire_bookshop_B
CREATE VIEW AdminService_Books_texts AS SELECT ... FROM sap_capire_booksh
CREATE VIEW CatalogService_Books_texts AS SELECT ... FROM sap_capire_book
CREATE VIEW localized_sap_capire_bookshop_Books AS SELECT ... FROM sap_ca
CREATE VIEW CatalogService_ListOfBooks AS SELECT ... FROM CatalogService_
CREATE VIEW localized_AdminService_Books AS SELECT ... FROM localized_sap
CREATE VIEW localized_CatalogService_Books AS SELECT ... FROM localized_s
CREATE VIEW localized_CatalogService_ListOfBooks AS SELECT ... FROM local
```

> **Note:** If you use SQLite, ALTER TYPE commands are not necessary and so, are not supported, as SQLite is essentially typeless.

Generate Scripts

You can use `cds deploy` with option `--script` to generate a script as a starting point for a manual migration. The effect of `--script` essentially is the same as for `--dry`, but it also allows changes that could lead to data loss and therefore are not supported in the automatic schema migration (see Limitations).

For generating such a script, perform the same steps as in section Dry-Run Offline above, but replace the command in step 3 by

```sh
cds deploy --script --delta-from cds-model.csn --out delta_script.sql
```

If your model change includes changes that could lead to data loss, there will be a warning and a respective comment is added to the dangerous statements in the resulting script. For example, deleting an element or reducing the length of an element would look like this:

delta_script.sql

```sql
...
-- [WARNING] this statement is lossy
ALTER TABLE sap_capire_bookshop_Books DROP price;

-- [WARNING] this statement could be lossy: length reduction of element "titl
ALTER TABLE sap_capire_bookshop_Books ALTER title TYPE VARCHAR(11);
...
```

WARNING

Always check and, if necessary, adapt the generated script before you apply it to your database!

Migration

Thanks to CAP's database-agnostic cds.ql API, we're confident that the new PostgreSQL service comes without breaking changes. Nevertheless, please check the instructions in the SQLite Migration guide, with by and large applies also to the new PostgreSQL service.

`cds deploy --model-only`

Not a breaking change, but definitely required to migrate former `cds-pg` databases, is to prepare it for schema evolution.

To do so run `cds deploy` once with the `--model-only` flag:

```sh
cds deploy --model-only
```

This will...:

- Create the `cds_model` table in your database.
- Fill it with the current model obtained through `cds compile '*'` .

> **IMPORTANT:**
>
> Your `.cds` models are expected to reflect the deployed state of your database.

With Deployer App

When you have a SaaS application, upgrade all your tenants using the deployer app with CLI option `--model-only` added to the start script command of your *package.json*. After having done that, don't forget to remove the `--model-only` option from the start script, to activate actual schema evolution.

MTX Support

> **WARNING**
>
> Multitenancy and extensibility aren't yet supported on PostgreSQL.